Server-Side ACTIONSCRIPT® Language Reference for ADOBE® MEDIA SERVER 5.0.1

Legal notices

Contents

Server-Side ActionScript Language Reference

Adobe Media Server server-side APIs 1
Global functions 1
Application class 6
ByteArray class 33
Client class 34
File class 55
GroupSpecifier class 71
GroupControl 78
LoadVars class 80
Log class 88
MulticastStreamInfo class 90
MulticastStreamIngest class 94
NetConnection class 96
NetGroup class 105
NetGroupInfo class 114
NetGroupReceiveMode class 115
NetGroupReplicationStrategy class 116
NetGroupSendMode class 117
NetGroupSendResult class 117
NetStream class 118
ProxyStream class 127
SharedObject class 131
SHA256 class 146
SOAPCall class 147
SOAPFault class 148
Stream class 150

Server-Side ActionScript Language Reference

Use Server-Side ActionScript™ to write server-side code for an Adobe® Media Server application. You can use Server-Side ActionScript to control login procedures, control events, communicate with other servers, allow and disallow users access to various server-side application resources, and let users update and share information.

Adobe Media Server server-side APIs

Server-Side ActionScript is Adobe's name for JavaScript 1.5. Adobe Media Server has an embedded Java-Script engine that compiles and executes server-side scripts. This *Server-Side ActionScript Language Reference* documents the Adobe Media Server host environment classes and functions. You can also use core Java-Script classes, functions, statements, and operators. For more information, see the Mozilla JavaScript documentation at http://developer.mozilla.org/en/JavaScript.

Server-Side ActionScript is similar, but not identical, to ActionScript 1.0. Both languages are based on ECMAScript (ECMA-262) edition 3 language specification. Server-Side ActionScript runs in the Mozilla SpiderMonkey engine embedded in Adobe Media Server. ActionScript 1.0 runs in AVM1 (ActionScript Virtual Machine 1) in Adobe® Flash® Player. SpiderMonkey implemented the ECMAScript specification exactly and Flash Player AVM1 did not. The biggest difference between Server-Side ActionScript and ActionScript 1.0 is that Server-Side ActionScript is case-sensitive.

Global functions

The following functions are available anywhere in a server-side script:

Signature	Description
`clearInterval()`	Stops a call to the `setInterval()` method.
`getGlobal()`	Provides access to the global object from the secure.asc file while the file is loading.
`load()`	Loads a Server-Side ActionScript file (ASC) or JavaScript file (JS) into the main.asc file.
`protectObject()`	Protects the methods of an object from application code.
`setAttributes()`	Prevents certain methods and properties from being enumerated, written, and deleted.
`setInterval()`	Calls a function or method at a specified time interval until the `clearInterval()` method is called.
`trace()`	Evaluates an expression and displays the value.

clearInterval()

```
clearInterval(intervalID)
```

Stops a call to the `setInterval()` method.

Availability
Flash Communication Server 1

Parameters
`intervalID` An identifier that contains the value returned by a previous call to the `setInterval()` method.

Example
The following example creates a function named `callback()` and passes it to the `setInterval()` method, which is called every 1000 milliseconds and outputs the message `"interval called."` The `setInterval()` method returns a number that is assigned to the `intervalID` variable. The identifier lets you cancel a specific `setInterval()` call. In the last line of code, the `intervalID` variable is passed to the `clearInterval()` method to cancel the `setInterval()` call.

```
function callback(){trace("interval called");}
var intervalID;
intervalID = setInterval(callback, 1000);
// sometime later
clearInterval(intervalID);
```

getGlobal()

`getGlobal()`

Provides access to the global object from the secure.asc file while the file is loading. Use the `getGlobal()` function to create protected system calls.

Availability
Flash Media Server 2

Details
Adobe Media Server has two script execution modes: secure and normal. In secure mode, only the secure.asc file (if it exists) is loaded and evaluated—no other application scripts are loaded. The `getGlobal()` and `protectObject()` functions are available only in secure mode. These functions are very powerful because they provide complete access to the script execution environment and let you create system objects. Once the secure.asc file is loaded, the server switches to normal script execution mode until the application is unloaded.

To prevent inadvertent access to the global object, always hold its reference in a temporary variable (declared by `var`); do not hold its reference in a member variable or a global variable.

Example
The following code gets a reference to the global object:

```
var global = getGlobal();
```

load()

`load(filename)`

Loads a Server-Side ActionScript file (ASC) or JavaScript file (JS) into the main.asc file. Call this function to load ActionScript libraries. The loaded file is compiled and executed after the main.asc file is successfully loaded, compiled, and executed, but before `application.onAppStart()` is called. The path of the specified file is resolved relative to the main.asc file.

Availability
Flash Communication Server 1

Parameters
`filename` A string indicating the relative path to a script file from the main.asc file.

Example
The following example loads the myLoadedFile.asc file:

```
load("myLoadedFile.asc");
```

protectObject()

```
protectObject(object)
```

Protects the methods of an object from application code. Application code cannot access or inspect the methods directly. You can use this function only in the secure.asc file.

Availability
Flash Media Server 2

Parameters
`object` An object to protect.

Returns
An Object.

Details
After an object is protected, don't reference it in global variables or make it a member of an accessible object. The object returned by `protectObject()` dispatches all method invocations to the underlying object but blocks access to member data. As a result, you can't enumerate or modify members directly. The protected object keeps an outstanding reference to the underlying object, which ensures that the object is valid. The protected object follows normal reference rules and exists while it is referred to.

Adobe Media Server has two script execution modes: secure and normal. In secure mode, only the secure.asc file (if it exists) is loaded and evaluated—no other application scripts are loaded. The `getGlobal()` and `protectObject()` functions are available only in secure mode. These functions are very powerful because they provide complete access to the script execution environment and let you create system objects. Once the secure.asc file is loaded, the server switches to normal script execution mode until the application is unloaded.

Example
After secure.asc is executed, calls to `load()` are directed through the user-defined system call, as shown in the following example:

```
var sysobj = {};
sysobj._load = load; // Hide the load function
load = null; // Make it unavailable unpriviliged code.
sysobj.load = function(fname){
    // User-defined code to validate/modify fname
    return this._load(fname);
}
// Grab the global object.
var global = getGlobal();

// Now protect sysobj and make it available as
// "system" globally. Also, set its attributes
// so that it is read-only and not deletable.

global["system"] = protectObject(sysobj);

setAttributes(global, "system", false, true, true);

// Now add a global load() function for compatibility.
// Make it read-only and nondeletable.

global["load"] = function(path){
    return system.load(path);
}

setAttributes(global, "load", false, true, true);
```

See also

LoadVars class

setAttributes()

```
setAttributes(object, propName, enumerable, readonly, permanent)
```

Prevents certain methods and properties from being enumerated, written, and deleted. In a server-side script, all properties in an object are enumerable, writable, and deletable by default. Call `setAttributes()` to change the default attributes of a property or to define constants.

Availability

Flash Media Server 2

Parameters

`object` An Object.

`propName` A string indicating the name of the property in the `object` parameter. Setting attributes on nonexistent properties has no effect.

`enumerable` One of the following values: `true`, `false`, or `null`. Makes a property enumerable if `true` or nonenumerable if `false`; a `null` value leaves this attribute unchanged. Nonenumerable properties are hidden from enumerations (`for var in` *obj*).

`readonly` One of the following values: `true`, `false`, or `null`. Makes a property read-only if `true` or writable if `false`; a `null` value leaves this attribute unchanged. Any attempt to assign a new value is ignored. Typically, you assign a value to a property while the property is writable and then make the property read-only.

permanent One of the following values: `true`, `false`, or `null`. Makes a property permanent (nondeletable) if `true` or deletable if `false`; a `null` value leaves this attribute unchanged. Any attempt to delete a permanent property (by calling `delete`*obj*`.`*prop*) is ignored.

Example
The following code prevents the `resolve()` method from appearing in enumerations:

```
Object.prototype.__resolve = function(methodName){ ... };
setAttributes(Object.prototype, "__resolve", false, null, null);
```

The following example creates three constants on a Constants object and makes them permanent and read-only:

```
Constants.KILO = 1000;
setAttributes(Constants, "KILO", null, true, true);
Constants.MEGA = 1000*Constants.KILO;
setAttributes(Constants, "MEGA", null, true, true);
Constants.GIGA = 1000*Constants.MEGA; setAttributes(Constants, "GIGA", null, true, true);
```

setInterval()

```
setInterval(function, interval[, p1, ..., pN])
setInterval(object.method, interval[, p1, ..., pN])
```

Calls a function or method at a specified time interval until the `clearInterval()` method is called. This method allows a server-side script to run a routine. The `setInterval()` method returns a unique ID that you can pass to the `clearInterval()` method to stop the routine.

Note: *Standard JavaScript supports an additional usage for the `setInterval()` method, `setInterval(stringToEvaluate, timeInterval)`, which is not supported by Server-Side ActionScript.*

Availability
Flash Communication Server 1

Parameters
`function` A Function object.

`object.method` A method to call on `object`.

`interval` A number indicating the time in milliseconds between calls to `function`.

`p1, ..., pN` Optional parameters passed to `function`.

Returns
An integer that provides a unique ID for this call. If the interval is not set, returns -1.

Example
The following example uses an anonymous function to send the message `"interval called"` to the server log every second:

```
setInterval(function(){trace("interval called");}, 1000);
```

The following example also uses an anonymous function to send the message “interval called” to the server log every second, but it passes the message to the function as a parameter:

```
setInterval(function(s){trace(s);}, 1000, "interval called");
```

The following example uses a named function, `callback1()`, to send the message `"interval called"` to the server log:

```
function callback1(){trace("interval called"); }
setInterval(callback1, 1000);
```

The following example also uses a named function, `callback2()`, to send the message `"interval called"` to the server log, but it passes the message to the function as a parameter:

```
function callback2(s){
    trace(s);
}
setInterval(callback2, 1000, "interval called");
```

The following example uses the second syntax:

```
var a = new Object();
a.displaying=displaying;
setInterval(a.displaying, 3000);

displaying = function(){
    trace("Hello World");
}
```

The previous example calls the `displaying()` method every 3 seconds and sends the message `"Hello World"` to the server log.

See also

`clearInterval()`

trace()

```
trace(expression)
```

Evaluates an expression and displays the value. You can use the `trace()` function to debug a script, to record programming notes, or to display messages while testing a file. The `trace()` function is similar to the `alert()` function in JavaScript.

The expression appears in the Live Log panel of the Administration Console; it is also published to the application.xx.log file located in a subdirectory of the *RootInstall*\logs folder. For example, if an application is called myVideoApp, the application log for the default application instance would be located here: *RootInstall*\logs_defaultVHost_\myVideoApp_definst_.

Availability

Flash Communication Server 1

Parameters

`expression` Any valid expression. The values in `expression` are converted to strings if possible.

Application class

Every instance of a Adobe Media Server application has an Application object, which is a single instance of the Application class. You don't need to use a constructor function to create an Application object; it is created automatically when an application is instantiated by the server.

Use the Application object to accept and reject client connection attempts, to register and unregister classes and proxies, and to manage the life cycle of an application. The Application object has callback functions that are invoked when an application starts and stops and when a client connects and disconnects.

Availability

Flash Communication Server 1

Property summary

Property	Description
`application.allowDebug`	A boolean value that lets administrators access an application with the Administration API `approveDebugSession()` method (`true`) or not (`false`).
`application.clients`	Read-only; an Array object containing a list of all the clients connected to an application.
`application.config`	Provides access to properties of the `ApplicationObject` element in the Application.xml configuration file.
`application.hostname`	Read-only; the host name of the server for default virtual hosts; the virtual host name for all other virtual hosts.
`application.name`	Read-only; the name of the application instance.
`application.server`	Read-only; the platform and version of the server.

Method summary

Method	Description
`application.acceptConnection()`	Accepts a connection call from a client to the server.
`application.broadcastMsg()`	Broadcasts a message to all clients connected to an application instance.
`application.clearSharedObjects()`	Deletes persistent shared objects files (FSO files) specified by the `soPath` parameter and clears all properties from active shared objects (persistent and nonpersistent).
`application.clearStreams()`	Clears recorded streams files associated with an application instance.
`application.denyPeerLookup()`	Specifies to the server that a peer lookup request has been denied.
`application.disconnect()`	Terminates a client connection to the application.
`application.gc()`	Invokes the garbage collector to reclaim any unused resources for this application instance.
`application.getStats()`	Returns statistics about an application.
`application.redirectConnection()`	Rejects a connection and provides a redirect URL.
`application.registerClass()`	Registers a constructor function that is used when deserializing an object of a certain class type.
`application.registerProxy()`	Maps a method call to another function.
`application.rejectConnection()`	Rejects the connection call from a client to the server.
`application.sendPeerRedirect()`	When a peer issues a lookup for a target peer, this method sends the peer an Array of addresses for the target peer.
`application.shutdown()`	Unloads the application instance.

Event handler summary

Event handler	Description
`application.onAppStart()`	Invoked when the server loads an application instance.
`application.onAppStop()`	Invoked when the server is about to unload an application instance.
`application.onConnect()`	Invoked when `NetConnection.connect()` is called from the client.
`application.onConnectAccept()`	Invoked when a client successfully connects to an application; for use with version 2 components only.
`application.onConnectReject()`	Invoked when a connection is rejected in an application that contains components.
`application.onDisconnect()`	Invoked when a client disconnects from an application.
`application.onPeerLookup()`	Invoked when the server receives a lookup request.
`application.onPublish()`	Invoked when a client publishes a stream to an application.
`application.onStatus()`	Invoked when the server encounters an error while processing a message that was targeted at this application instance.
`application.onUnpublish()`	Invoked when a client stops publishing a stream to an application.

application.acceptConnection()

`application.acceptConnection(clientObj)`

Accepts a connection call from a client to the server.

Availability

Flash Communication Server 1

Parameters

`clientObj` A Client object; a client to accept.

Details

When `NetConnection.connect()` is called from the client side, it passes a Client object to `application.onConnect()` on the server. Call `application.acceptConnection()` in an `application.onConnect()` event handler to accept a connection from a client. When this method is called, `NetConnection.onStatus()` is invoked on the client with the `info.code` property set to `"NetConnection.Connect.Success"`.

You can use the `application.acceptConnection()` method outside an `application.onConnect()` event handler to accept a client connection that had been placed in a pending state (for example, to verify a user name and password).

When you call this method, `NetConnection.onStatus()` is invoked on the client with the `info.code` property set to `"NetConnection.Connect.Success"`. For more information, see the `NetStatusEvent.info` property in the *ActionScript 3.0 Language and Components Reference* or the `NetConnection.onStatus()` entry in the *Adobe Media Server ActionScript 2.0 Language Reference*.

Note: *When you use version 2 components, the last line (in order of execution) of the `onConnect()` handler should be either `application.acceptConnection()` or `application.rejectConnection()` (unless you're leaving the application in a pending state). Also, any logic that follows `acceptConnection()` or `rejectConnection()` must be placed in the `application.onConnectAccept()` and `application.onConnectReject()` handlers, or it will be ignored.*

Example

The following server-side code accepts a client connection and traces the client ID:

```
application.onConnect = function(client){
    // Accept the connection.
        application.acceptConnection(client);
    trace("connect: " + client.id);
};
```

Note: *This example shows code from an application that does not use components.*

application.allowDebug

```
application.allowDebug
```

A boolean value that lets administrators access an application with the Administration API `approveDebugSession()` method (`true`) or not (`false`). A debug connection lets administrators view information about shared objects and streams in the Administration Console.

The default value for this property is `false` and is set in the Application.xml file:

```
<Application>
    ...
    <Debug>
        <AllowDebugDefault>false</AllowDebugDefault>
    </Debug>
    ...
</Application>
```

Setting `application.allowDebug` to `true` in a server-side script overrides the value in the Application.xml file. To view information in the Administration Console about the shared objects and streams in an application, add the following line to your code:

```
application.allowDebug = true;
```

Availability

Flash Media Server 2

application.broadcastMsg()

```
application.broadcastMsg(cmd [, p1,..., pN])
```

Broadcasts a message to all clients connected to an application instance. To handle the message, the client must define a handler on the NetConnection object with the same name as the `cmd` parameter.

In ActionScript 2.0, define the method on the NetConnection object. In ActionScript 3.0, assign the `NetConnection.client` property to an object on which callback methods are invoked. Define the method on that object.

Availability

Flash Media Server 2

Parameters

`cmd` A string; the name of the a handler on the client-side NetConnection object.

`p1,..., pN` A string; messages to broadcast.

Example

The following server-side code sends a message to the client:

```
application.broadcastMsg("serverMessage", "Hello Client");
```

The following client-side ActionScript 2.0 code handles the message and outputs “Hello Client”:

```
nc = new NetConnection();
nc.serverMessage = function(msg){
    trace(msg);
};
```

The following client-side ActionScript 3.0 code handles the message and outputs “Hello Client”:

```
var nc:NetConnection = new NetConnection()
var ncClient = new Object();
nc.client = ncClient;
ncClient.serverMessage = nc_serverMessage;
function nc_serverMessage(msg:String):void{
    trace(msg);
}
```

application.clearSharedObjects()

```
application.clearSharedObjects(soPath)
```

Deletes persistent shared objects files (FSO files) specified by the `soPath` parameter and clears all properties from active shared objects (persistent and nonpersistent). Even if you have deleted all the properties from a persistent shared object, unless you call `clearSharedObjects()`, the FSO file still exists on the server.

Availability

Flash Communication Server 1

Parameters

`soPath` A string indicating the Uniform Resource Identifier (URI) of a shared object.

The `soPath` parameter specifies the name of a shared object, which can include a slash (/) as a delimiter between directories in the path. The last element in the path can contain wildcard patterns (for example, a question mark [?] and an asterisk [*]) or a shared object name. The `application.clearSharedObjects()` method traverses the shared object hierarchy along the specified path and clears all the shared objects. Specifying a slash (/) clears all the shared objects that are associated with an application instance.

If `soPath` matches a shared object that is currently active, all its properties are deleted, and a `clear` event is sent to all subscribers of the shared object. If it is a persistent shared object, the persistent store is also cleared.

The following values are possible for the `soPath` parameter:

- `/` clears all local and persistent shared objects associated with the instance.
- `/foo/bar` clears the shared object `/foo/bar`; if bar is a directory name, no shared objects are deleted.
- `/foo/bar/*` clears all shared objects stored under the instance directory /foo/bar. If no persistent shared objects are in use within this namespace, the bar directory is also deleted.
- `/foo/bar/XX??` clears all shared objects that begin with `XX`, followed by any two characters. If a directory name matches this specification, all the shared objects within this directory are cleared.

Returns

A boolean value of `true` if the shared object at the specified path was deleted; otherwise, `false`. If wildcard characters are used to delete multiple files, the method returns `true` only if all the shared objects that match the wildcard pattern were successfully deleted; otherwise, it returns `false`.

Example

The following example clears all the shared objects for an instance:

```
function onApplicationStop(){
    application.clearSharedObjects("/");
}
```

application.clearStreams()

```
application.clearStreams(streamPath)
```

Clears recorded streams files associated with an application instance. You can use this method to clear a single stream, all streams associated with the application instance, just those streams in a specific subdirectory of the application instance, or just those streams whose names match a specified wildcard pattern.

If the `clearStreams()` method is invoked on a stream that is currently recording, the recorded file is set to length 0 (cleared), and the internal cached data is also cleared.

A call to `application.clearStreams()` invokes the `Stream.onStatus()` handler and passes it an information object that contains information about the success or failure of the call.

Note: *You can also use the Administration API `removeApp()` method to delete all the resources for a single application instance.*

Availability

Flash Communication Server 1

Parameters

`streamPath` A string indicating the Uniform Resource Identifier (URI) of a stream.

The `streamPath` parameter specifies the location and name of a stream relative to the directory of the application instance. You can include a slash (/) as a delimiter between directories in the path. The last element in the path can contain wildcard patterns (for example, a question mark [?] and an asterisk [*]) or a stream name. The `clearStreams()` method traverses the stream hierarchy along the specified path and clears all the recorded streams that match the given wildcard pattern. Specifying a slash clears all the streams that are associated with an application instance.

To clear FLV, F4V, or MP3 files, precede the stream path with `flv:`, `mp4:`, or `mp3:`. When you specify `flv:` or `mp3:` you don't have to specify a file extension; .flv and .mp3 are implied. However, when you call `application.clearStreams("mp4:foo")`, the server deletes any file with the name "foo" in an MPEG-4 container; for example, foo.mp4, foo.mov, and foo.f4v. To delete a specific file, pass the file extension in the call; for example, `application.clearStreams("mp4:foo.f4v")`.

Note: *If you don't precede the stream path with a file type, only FLV files are deleted.*

The following examples show some possible values for the `streamPath` parameter:

- `flv:/` clears all FLV streams associated with the application instance.
- `mp3:/` clears all MP3 files associated with the application instance.

- `mp4:/` clears all F4V streams associated with the application instance (for example, foo.mp4, foo.f4v, and so on).
- `mp4:foo.mp4` clears the foo.mp4 file.
- `mp4:foo.mov` clears the foo.mov file.
- `mp3:/mozart/requiem` clears the MP3 file named requiem.mp3 from the application instance's /mozart subdirectory.
- `mp3:/mozart/*` clears all MP3 files from the application instance's /mozart subdirectory.
- `/report` clears the report.flv stream file from the application instance directory.
- `/presentations/intro` clears the recorded intro.flv stream file from the application instance's /presentations subdirectory; if `intro` is a directory name, no streams are deleted.
- `/presentations/*` clears all FLV files from the application instance's /presentations subdirectory. The /presentation subdirectory is also deleted if no streams are used in this namespace.
- `/presentations/report??` clears all FLV files that begin with "report," followed by any two characters. If there are directories within the given directory listing, the directories are cleared of any streams that match `report??`.

Returns

A boolean value of `true` if the stream at the specified path was deleted; otherwise, `false`. If wildcard characters are used to clear multiple stream files, the method returns `true` only if all the streams that match the wildcard pattern were successfully deleted; otherwise, it returns `false`.

Example

The following example clears all recorded streams:

```
function onApplicationStop(){
    application.clearStreams("/");
}
```

The following example clears all MP3 files from the application instance's /disco subdirectory:

```
function onApplicationStop(){
    application.clearStreams("mp3:/disco/*");
}
```

Removing all HDS segments

To remove all the existing HDS segments when the application unloads, you can use the `clearOnAppStop` tag as shown below:

```
<JSEngine>
            <ApplicationObject>
                <config>
                    <clearOnAppStop>true</clearOnAppStop></config>
            </ApplicationObject>
</JSEngine>
```

application.clients

```
application.clients
```

Read-only; an Array object containing a list of all the clients connected to an application. Each element in the array is a reference to the Client object; use the `application.clients.length` property to determine the number of users connected to an application.

Do not use the index value of the `clients` array to identify users between calls, because the array is compacted when users disconnect and the slots are reused by other Client objects.

Availability
Flash Communication Server 1

Example
The following example uses a `for` loop to iterate through each element in the `application.clients` array and calls the `serverUpdate()` method on each client:

```
for (i = 0; i < application.clients.length; i++){
    application.clients[i].call("serverUpdate");
}
```

application.config

`application.config`

Provides access to properties of the `ApplicationObject` element in the Application.xml configuration file. To access properties that you set in the configuration file, use the `application.config` property. For example, to set the value of the `password` element, use the code `application.config.password`.

Availability
Flash Media Server 2

Example
Use this sample section from an Application.xml file for this example:

```
<Application>
    <ScriptEngine>
            <ApplicationObject>
                <config>
                <user_name>jdoe</user_name>
                <dept_name>engineering</dept_name>
            </config>
            </ApplicationObject>
    </ScriptEngine>
</Application>
```

Note: *You must use the <ScriptEngine> tag as <JSEngine> tag is deprecated.*

The following lines of code access the `user_name` and `dept_name` properties:

```
trace("I am " + application.config.user_name + " and I work in the " +
application.config.dept_name + " department.");

trace("I am " + application.config["user_name"] + " and I work in the " +
application.config["dept_name"] + " department.");
```

The following code is sent to the application log file and the Administration Console:

```
I am jdoe and I work in the engineering department.
```

application.denyPeerLookup()

`application.denyPeerLookup(tag:ByteArray)`

Specifies to the server that a peer lookup request has been denied. A call to `denyPeerLookup()` increments the `rtmfp_lookups_deny` statistic in the `getServerStats()` Administration API. This call also logs a message specifying the lookup parameters.

For more information, see Filter introduction requests in the *Adobe Media Server Developer's Guide*.

Availability
Flash Media Server 4.5

Parameters
`tag` ByteArray; The `event.tag` received in the lookup request.

Returns
Nothing.

application.disconnect()

`application.disconnect(clientObj)`

Terminates a client connection to the application. When this method is called, `NetConnection.onStatus()` is invoked on the client with `info.code` set to `"NetConnection.Connect.Closed"`. The `application.onDisconnect()` handler is also invoked.

Availability
Flash Communication Server 1

Parameters
`clientObj` A Client object indicating the client to disconnect. The object must be a Client object from the `application.clients` array.

Returns
A boolean value of `true` if the disconnection was successful; otherwise, `false`.

Example
The following example calls `application.disconnect()` to disconnect all users from an application instance:

```
function disconnectAll(){
    for (i=0; i < application.clients.length; i++){
        application.disconnect(application.clients[i]);
    }
}
```

application.gc()

`application.gc()`

Invokes the garbage collector to reclaim any unused resources for this application instance.

Availability
Flash Media Server 2

application.getStats()

`application.getStats()`

Returns statistics about an application.

Availability

Flash Communication Server 1

Returns

An Object whose properties contain statistics about the application instance. The following table describes the properties:

Property	Description
`bw_in`	Total number of kilobytes received.
`bw_out`	Total number of kilobytes sent.
`bytes_in`	Total number of bytes sent.
`bytes_out`	Total number of bytes received. Note: For billing, use the `sc-bytes` field in the Access log.
`msg_in`	Total number of Real-Time Messaging Protocol (RTMP) messages sent.
`msg_out`	Total number of RTMP messages received.
`msg_dropped`	Total number of RTMP messages dropped.
`server_bytes_in`	Total number of bytes received by the server.
`server_bytes_out`	Total number of bytes sent by the server.
`total_connects`	Total number of clients connected to an application instance.
`total_disconnects`	Total number of clients who have disconnected from an application instance.

Example

The following example outputs application statistics to the Live Log panel in the Administration Console:

```
function testStats(){
    var stats = application.getStats();
    for(var prop in stats){
         trace("stats." + prop + " = " + stats[prop]);
    }
}

application.onConnect = function(client){
    this.acceptConnection(client);
    testStats();
};
```

application.hostname

`application.hostname`

Read-only; the host name of the server for default virtual hosts; the virtual host name for all other virtual hosts.

If an application is running on the default virtual host, and if a value is set in the `ServerDomain` element in the Server.xml configuration file, the `application.hostname` property contains the value set in the `ServerDomain` element. If a value has not been set in the `ServerDomain` element, the property is undefined.

If an application is running on any virtual host other than the default, the `application.hostname` property contains the name of the virtual host.

Availability
Flash Communication Server 1.5

application.name

`application.name`

Read-only; the name of the application instance.

Availability
Flash Communication Server 1

Example
The following example checks the `name` property against a specific string before it executes some code:

```
if (application.name == "videomail/work"){
    // Insert code here.
}
```

application.onAppStart()

`application.onAppStart = function (){}`

Invoked when the server first loads the application instance. Use this handler to initialize an application state. The `onAppStart()` event is invoked only once during the lifetime of an application instance.

Availability
Flash Communication Server 1

Example

```
application.onAppStart = function (){
    trace ("*** sample_guestbook application start");

    // Create a reference to a persistent shared object.
    application.entries_so = SharedObject.get("entries_so", true);

    // Prevent clients from updating the shared object.
    application.entries_so.lock();

    // Get the number of entries saved in the shared object
    // and save it in application.lastEntry.
    var maxprop = 0;
    var soProperties = application.entries_so.getPropertyNames();
    trace("soProperties:" + soProperties);
    if (soProperties == null) {
        application.lastEntry = 0;
    } else {
        for (var prop in soProperties) {
            maxprop = Math.max (parseInt(prop), maxprop);
            trace("maxprop " + maxprop);
        }
        application.lastEntry = maxprop+1;
    }
    // Allow clients to update the shared object.
    application.entries_so.unlock();
    trace("*** onAppStart called.");
};
```

application.onAppStop()

```
application.onAppStop = function (info){}
```

Invoked when the server is about to unload an application instance. You can use `onAppStop()` to flush the application state or to prevent the application from being unloaded.

Define a function that is executed when the event handler is invoked. If the function returns `true`, the application is unloaded. If the function returns `false`, the application is not unloaded. If you don't define a function for this event handler, or if the return value is not a boolean value, the application is unloaded when the event is invoked.

The Adobe Media Server application passes an information object to the `application.onAppStop()` event. You can use Server-Side ActionScript to look at this information object to decide what to do in the function you define. You can also use the `application.onAppStop()` event to notify users before shutdown.

If you use the Administration Console or the Server Administration API to unload a Adobe Media Server application, `application.onAppStop()` is not invoked. Therefore you cannot use `application.onAppStop()` to tell users that the application is exiting.

When an application doesn't have incoming client connections, the server considers the application idle and unloads it. To prevent this, define an `Application.onAppStop()` handler that returns `false`.

Availability

Flash Communication Server 1

Parameters

`info` An Object, called an *information object*, with properties that explain why the application is about to stop running. The information object has a `code` property and a `level` property.

Code property	Level property	Description
`Application.Shutdown`	`status`	The application instance is about to shut down.
`Application.GC`	`status`	The application instance is about to be destroyed by the server.

Returns

The value returned by the function you define, if any, or `null`. To unload the application, return `true` or any non-`false` value. To refuse to unload the application, return `false`.

Example

The following example flushes the `entries_so` shared object when the application stops:

```
application.onAppStop = function (info){
    trace("*** onAppStop called.");
    if (info=="Application.Shutdown"){
        application.entries_so.flush();
    }
}
```

application.onConnect()

```
application.onConnect = function (clientObj [, p1, ..., pN]){}
```

Invoked when `NetConnection.connect()` is called from the client. This handler is passed a Client object representing the connecting client. Use the Client object to perform actions on the client in the handler. For example, use this function to accept, reject, or redirect a client connection, perform authentication, define methods on the Client object to be called remotely from `NetConnection.call()`, and set the `Client.readAccess` and `Client.writeAccess` properties to determine client access rights to server-side objects.

When performing authentication, all of the information required for authentication should be sent from the `NetConnection.connect()` method to the `onConnect()` handler as parameters (*p1*..., *pN*).

If you don't define an `onConnect()` handler, connections are accepted by default.

If there are several simultaneous connection requests for an application, the server serializes the requests so that only one `application.onConnect()` handler is executed at a time. It's a good idea to write code for the `application.onConnect()` function that is executed quickly to prevent a long connection time for clients.

Note: *When you are using the version 2 component framework (that is, when you are loading the components.asc file in your server-side script file), you must use the `application.onConnectAccept()` method to accept client connections.*

Availability

Flash Communication Server 1

Parameters

`clientObj` A Client object. This object contains information about the client that is connecting to the application.

`p1 ..., pN` Optional parameters passed to the `application.onConnect()` handler from the client-side `NetConnection.connect()` method when a client connects to the application.

Returns

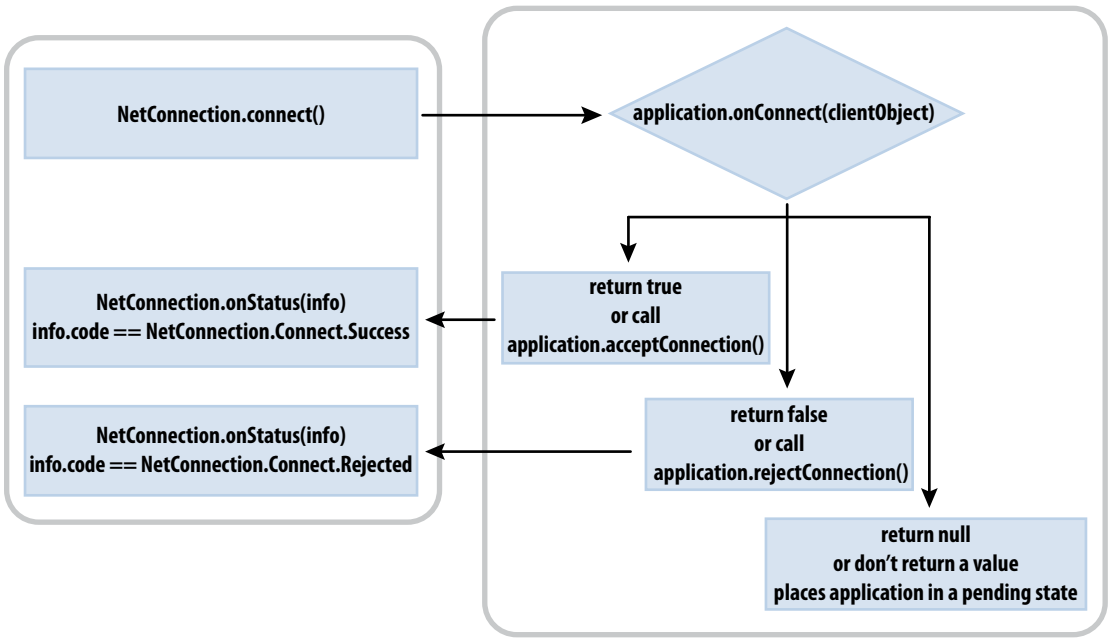
A boolean value; `true` causes the server to accept the connection; `false` causes the server to reject the connection.

When `true` is returned, `NetConnection.onStatus()` is invoked on the client with `info.code` set to `"NetConnection.Connect.Success"`. When `false` is returned, `NetConnection.onStatus()` is invoked on the client with `info.code` set to `"NetConnection.Connect.Rejected"`.

If `null` or no value is returned, the server puts the client in a pending state and the client can't receive or send messages. If the client is put in a pending state, you must call `application.acceptConnection()` or `application.rejectConnection()` at a later time to accept or reject the connection. For example, you can perform external authentication by making a NetConnection call in your `application.onConnect()` event handler to an application server and having the reply handler call `application.acceptConnection()` or `application.rejectConnection()`, depending on the information received by the reply handler.

You can also call `application.acceptConnection()` or `application.rejectConnection()` in the `application.onConnect()` event handler. If you do, any value returned by the function is ignored.

Note: *Returning 1 or 0 is not the same as returning `true` or `false`. The values 1 and 0 are treated the same as any other integers and do not accept or reject a connection.*



A. *Client-side ActionScript* ***B.*** *Server-Side ActionScript*

Example

The following examples show three ways to accept or reject a connection in the `onConnect()` handler:

```
(Usage 1)
application.onConnect = function (clientObj [, p1, ..., pN]){
    // Insert code here to call methods that do authentication.
    // Returning null puts the client in a pending state.
    return null;
};
(Usage 2)
application.onConnect = function (clientObj [, p1, ..., pN]){
    // Insert code here to call methods that do authentication.
    // The following code accepts the connection:
    application.acceptConnection(clientObj);
};
(Usage 3)
application.onConnect = function (clientObj [, p1, ..., pN])
{
    // Insert code here to call methods that do authentication.
    // The following code accepts the connection by returning true:
    return true;
};
```

The following example verifies that the user has sent the password "XXXX". If the password is sent, the user's access rights are modified and the user can complete the connection. In this case, the user can create or write to streams and shared objects in the user's own directory and can read or view any shared object or stream in this application instance.

```
// This code should be placed in the global scope.

application.onConnect = function (newClient, userName, password){
    // Do all the application-specific connect logic.
    if (password == "XXXX"){
        newClient.writeAccess = "/" + userName;
        this.acceptConnection(newClient);
    } else {
        var err = new Object();
        err.message = "Invalid password";
        this.rejectConnection(newClient, err);
    }
};
```

If the password is incorrect, the user is rejected and an information object with a `message` property set to `"Invalid password"` is returned to the client side. The object is assigned to `infoObject.application`. To access the `message` property, use the following code on the client side:

```
ClientCom.onStatus = function (info.application.message){
    trace(info.application.message);
    // Prints "Invalid password"
    // in the Output panel on the client side.
};
```

application.onConnectAccept()

```
application.onConnectAccept = function (clientObj [,p1, ..., pN]){}
```

Invoked when a client successfully connects to an application; for use with version 2 components only. Use `onConnectAccept()` to handle the result of an accepted connection in an application that contains components.

Note: *This component set is deprecated and not included with Adobe Media Server versions 4.0 and later. The components are available from www.adobe.com/go/ams_tools.*

If you don't use the version 2 components framework (ActionScript 2.0 components), you can execute code in the `application.onConnect()` handler after accepting or rejecting the connection. When you use the components framework, however, any code that you want to execute after the connection is accepted or rejected must be placed in the `application.onConnectAccept()` and `application.onConnectReject()` event handlers. This architecture allows all of the components to decide whether a connection is accepted or rejected.

Availability

Flash Media Server (with version 2 media components only).

Parameters

`clientObj` A Client object; the client connecting to the application.

`p1, ..., pN` Optional parameters passed to the `application.onConnectAccept()` method. These parameters are passed from the client-side `NetConnection.connect()` method when a client connects to the application; they can be any ActionScript data type.

Example

The following example is client-side code:

```
nc = new NetConnection();
nc.connect("rtmp:/test","jlopes");

nc.onStatus = function(info) {
    trace(info.code);
};

nc.doSomething = function(){
    trace("doSomething called!");
}
```

The following example is server-side code:

```
// When using components, always load components.asc.
load("components.asc");

application.onConnect = function(client, username){
    trace("onConnect called");
        gFrameworkFC.getClientGlobals(client).username = username;
    if (username == "hacker") {
        application.rejectConnection(client);
    }
    else {
        application.acceptConnection(client);
    }
}

// Code is in onConnectAccept and onConnectReject statements
// because components are used.
application.onConnectAccept = function(client, username){
    trace("Connection accepted for "+username);
    client.call("doSomething",null);
}

application.onConnectReject = function(client, username){
    trace("Connection rejected for "+username);
}
```

application.onConnectReject()

```
application.onConnectReject = function (clientObj [,p1, ..., pN]){}
```

Invoked when a connection is rejected in an application that contains components.

Note: *This component set is deprecated and not included with Flash Media Server versions 4.0 and later. The components are available from www.adobe.com/go/ams_tools.*

If you don't use the version 2 components framework, you can execute code in the `application.onConnect()` handler after accepting or rejecting a connection. When you use the components framework, however, any code that you want to execute after the connection is accepted or rejected must be placed in the `application.onConnectAccept()` and `application.onConnectReject()` framework event handlers. This architecture allows all of the components to decide whether a connection is accepted or rejected.

Availability

Flash Media Server (with version 2 components only)

Parameters

`clientObj` A Client object; the client connecting to the application.

`p1, ..., pN` Optional parameters passed to the `application.onConnectReject()` handler. These parameters are passed from the client-side `NetConnection.connect()` method when a client connects to the application.

Example

The following example is client-side code that you can use for an application:

```
nc = new NetConnection();
nc.connect("rtmp:/test","jlopes");

nc.onStatus = function(info) {
    trace(info.code);
};

nc.doSomething = function(){
    trace("doSomething called!");
}
```

The following example is server-side code that you can include in the main.asc file:

```
// When using components, always load components.asc.
load( "components.asc" );

application.onConnect = function(client, username){
    trace("onConnect called");
        gFrameworkFC.getClientGlobals(client).username = username;
    if (username == "hacker") {
        application.rejectConnection(client);
    }
    else {
        application.acceptConnection(client);
    }
}

application.onConnectAccept = function(client, username){
    trace("Connection accepted for "+username);
    client.call("doSomething",null);
}

application.onConnectReject = function(client, username){
    trace("Connection rejected for "+username);
}
```

application.onDisconnect()

```
application.onDisconnect = function (clientObj){}
```

Invoked when a client disconnects from an application. Use this event handler to flush any client state information or to notify other users that a user is leaving the application. This handler is optional.

Note: *After a client has disconnected from an application, you cannot use this method to send data back to that disconnected client.*

Availability

Flash Communication Server 1

Parameters

`clientObj` A Client object; a client disconnecting from the application.

Returns

Server ignores any return value.

Example

This example notifies all connected clients when a client disconnects from an application. The client-side FLA file contains an input text field called `nameText`, a dynamic text field called `statusText`, and a button called `connectButton`. The user enters their name in the input text field. The client-side code passes the name to the server in the `NetConnection.connect()` call, as follows:

```
nc = new NetConnection();
nc.userDisconnects = function(name) {
    statusText.text = name + ": disconnected";
}
nc.onStatus = function(info){
    statusText.text = info.code;
}
connectButton.onPress = function() {
    nc.connect("rtmp://localhost/testapp", nameText.text);
};
```

The server-side `onConnect()` handler receives the user name from the client-side code and assigns it to a property of the Client object. The server passes the Client object to the `onDisconnect()` handler when a client disconnects from the application. The `Client.call()` method inside the `onDisconnect()` handler calls the `userDisconnects` method on the client and passes it the name of the disconnecting client. The client displays the name of the disconnected user.

```
application.onConnect = function(client, name){
    client.name = name;
    trace(client.name + ": onConnect");
    return true;
}
application.onDisconnect = function(client){
    for (var i = 0; i < application.clients.length; i++){
        application.clients[i].call("userDisconnects", null, client.name);
    }
    trace(client.name + ": onDisconnect");
}
```

Note: *To pass optional parameters to the `Client.call()` method, pass `null` for the second (`responseObject`) parameter.*

application.onPeerLookup()

```
application.onPeerLookup = function (event:Object){}
```

Invoked when Adobe Media Server receives a request from a client to connect to another peer. The *initiating peer* is the peer that makes the request. The *target peer* is the peer to which the initiating peer wants to connect. Both the initiating peer and the target peer can be connected directly to the server or remote. To reply to the peer lookup request, call `application.sendPeerRedirect()`.

Availability

Flash Media Server 4.5

Parameters

`event` Object; contains information about the intiating peer and the target peer. Pass the `event` object in calls to `application.sendPeerRedirect()`. Pass the `event.tag` object in calls to `Client.introducePeer()`.

The `event` object contains the following properties:

Property	Data type	Description
`targetPeerID`	String	The peerID of the target peer. The initiating peer is attempting to look up and connect to the target peer.
`initiatorAddress`	String	The IP address of the initiating client. Send redirect information to this address.
`tag`	ByteArray	A value that uniquely identifies this lookup request.
`interfaceID`	Number	Identifies the RTMFP interface on which the request was received.

Example

For an example of distributing peer lookup requests across multiple servers, see Distribute peer lookup requests across multiple servers.

See also

`application.sendPeerRedirect()`, `Client.introducePeer()`

application.onPublish()

```
application.onPublish = function (clientObj, streamObj){}
```

Invoked when a client publishes a stream to an application. Use this event handler to send traffic to other servers when you're building a large-scale live broadcasting application; this is called *multipoint publishing*. For example, you can support subscribers in multiple geographic locations by sending traffic from the origin server (Server A) in one city to two origin servers in two different cities (Server B and Server C). The following is the workflow for such a scenario:

1. A client publisher connects to Server A and starts publishing.
2. Server A receives notifications from the event handler `application.onPublish()` in a server-side script.
3. Inside the `onPublish()` handler, create two NetStream objects to Server B and Server C.
4. Call the `NetStream.publish()` method to redirect the publishing data from Server A to Server B and Server C.
5. Subscribers connecting to Server B and Server C get the same live stream.

In this example, the publishing client connects and publishes only to Server A. The rest of the data flow is handled by logic in the server-side script.

Note: *You cannot change Client object properties in this handler.*

Availability

Flash Media Server 3

Parameters

`clientObj` A Client object; the client publishing the stream to the application.

`streamObj` A Stream object; the stream being published to the application.

Returns

Server ignores any return value.

application.onStatus()

```
application.onStatus = function (infoObject){}
```

Invoked when the server encounters an error while processing a message that was targeted at this application instance. The `application.onStatus()` handler handles any `Stream.onStatus()` or `NetConnection.onStatus()` messages that don't find handlers. Also, there are a few status calls that come only to `application.onStatus()`.

Availability
Flash Communication Server 1

Parameters
`infoObject` An Object with `code` and `level` properties that contain information about the status of an application. Some information objects also have `details` and `description` properties.The following table describes the information object property values:

Code property	Level property	Description
`Application.Script.Error`	`error`	The ActionScript engine has encountered a runtime error. This information object also has the following properties: • `filename`: name of the offending ASC file. • `lineno`: line number where the error occurred. • `linebuf`: source code of the offending line.
`Application.Script.Warning`	`warning`	The ActionScript engine has encountered a runtime warning. This information object also has the following properties: • `filename`: name of the offending ASC file. • `lineno`: line number where the error occurred. • `linebuf`: source code of the offending line.
`Application.Resource.LowMemory`	`warning`	The ActionScript engine is low on runtime memory. This provides an opportunity for the application instance to free some resources or to take suitable action. If the application instance runs out of memory, it is unloaded and all users are disconnected. In this state, the server does not invoke the `application.onDisconnect()` event handler or the `application.onAppStop()` event handler.

Returns
Any value that the callback function returns.

Example
```
application.onStatus = function(info){
    trace("code: " + info.code + " level: " + info.level);
    trace(info.code + " details: " + info.details);
};
// Application.Script.Warning level: warning
```

application.onUnpublish()

```
application.onUnpublish = function (clientObj, streamObj){}
```

Invoked when a client stops publishing a stream to an application. Use this event handler with `application.onPublish()` to send traffic to other servers when you're building a large-scale, live broadcasting application.

Note: *You cannot change Client object properties in this handler.*

Availability
Flash Media Server 3

Parameters
`clientObj` A Client object; the client publishing the stream to the application.

`streamObj` A Stream object; the stream being published to the application.

Returns
Server ignores any return value.

application.redirectConnection()

```
application.redirectConnection(clientObj, url[, description[, errorObj]])
```

Rejects a connection and provides a redirect URL. You must write logic in the `NetConnection.onStatus()` handler that detects redirection and passes the new connection URL to the `NetConnection.connect()` method.

When this method is called, `NetConnection.onStatus()` is invoked on the client and passed an information object with the following values:

Property	Value
`info.code`	`"NetConnection.Connect.Rejected"`
`info.description`	The value passed in the description parameter; if no value is passed in the parameter, the default value is `"Connection failed"`
`info.ex.code`	`302`
`info.ex.redirect`	The new connection URL
`info.level`	`"Error"`

Availability
Flash Media Server 3

Parameters
`clientObj` A Client object specifying a client to reject.

`url` A string specifying the new connection URL.

Note: *If you omit this parameter, `rejectConnection()` is called instead.*

`description` A string that lets you provide more information when a connection is redirected.

`errorObj` An object of any type that is sent to the client, explaining the reason for rejection. The `errorObj` object is available in client-side scripts as the `application` property of the information object that is passed to the `NetConnection.onStatus()` call when the connection is rejected.

Example

The following example is server-side code:

```
application.onConnect = function(clientObj, count){
    var err = new Object();
    err.message = "This is being rejected";
    err.message2 = "This is the second message. with number description";
    if (count == 1){
        redirectURI = "rtmp://www.example.com/redirected/fromScript";
        redirectDescription = "this is being rejected via Server Side Script.";
    }
    else if (count == 2){
        redirectURI = "rtmp://www.example2.com/redirected/fromScript";
        redirectDescription = "this is being rejected via Server Side Script.";
    }
    application.redirectConnection(clientObj, redirectURI, redirectDescription, err);
}
```

The following example is client-side ActionScript 3.0 code:

```
var theConnection:NetConnection;
var theConnection2:NetConnection;
var client:Object = new Object();

function init():void{
    connect_button.label = "Connect";
    disconnect_button.label = "Disconnect";

    connect_button.addEventListener(MouseEvent.CLICK, buttonHandler);
    disconnect_button.addEventListener(MouseEvent.CLICK, buttonHandler);
}

function buttonHandler(event:MouseEvent){
    switch (event.target){
        case connect_button :
            doConnect();
            break;
        case disconnect_button :
            disConnect();
            break;
    }
}
function doConnect(){
    makeConnection(theURI.text);
}


function disConnect(){
    theConnection.close();
}to

function makeConnection(uri:String){
    if (theConnection){
        theConnection.close();
    }
    theConnection = new NetConnection();
    theConnection.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
```

```
    theConnection.client = client;
    theConnection.connect(uri);
}

function makeConnection2(uri:String){
    if (theConnection2){
        theConnection2.close();
    }
    theConnection2 = new NetConnection();
    theConnection2.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    theConnection2.client = client;
    theConnection2.connect(uri);
}

function netStatusHandler(event:NetStatusEvent):void{
//Check the Redirect code and make connection to redirect URI if appropriate.
    try{
        if (event.info.ex.code == 302){
            var redirectURI:String;
            redirectURI = event.info.ex.redirect;
            if (redirectURI.charCodeAt(redirectURI.length-1) == 13){
                redirectURI = redirectURI.slice(0,(redirectURI.length-1));
            }
            makeConnection2(redirectURI);
        }
    }
}

init();
```

application.registerClass()

`application.registerClass(className, constructor)`

Registers a constructor function that is used when deserializing an object of a certain class type. If the constructor for a class is not registered, you cannot call the deserialized object's methods. This method is also used to unregister the constructor for a class. This is an advanced use of the server and is necessary only when sending ActionScript objects between a client and a server.

The client and the server communicate over a network connection. Therefore, if you use typed objects, each side must have the prototype of the same objects they both use. In other words, both the client-side and Server-Side ActionScript must define and declare the types of data they share so that there is a clear, reciprocal relationship between an object, method, or property on the client and the corresponding element on the server. You can call `application.registerClass()` to register the object's class type on the server side so that you can use the methods defined in the class.

Constructor functions should be used to initialize properties and methods; they should not be used for executing server code. Constructor functions are called automatically when messages are received from the client and need to be "safe" in case they are executed by a malicious client. You shouldn't define procedures that could result in negative situations, such as filling up the hard disk or consuming the processor.

The constructor function is called before the object's properties are set. A class can define an `onInitialize()` method, which is called after the object has been initialized with all its properties. You can use this method to process data after an object is deserialized.

If you register a class that has its prototype set to another class, you must set the prototype constructor back to the original class after setting the prototype. The second example below illustrates this point.

Note: *Client-side classes must be defined as* `function function_name(){}`*, as shown in the following examples. If not defined in the correct way,* `application.registerClass()` *does not identify the class when its instance passes from the client to the server, and an error is returned.*

Availability

Flash Communication Server 1

Parameters

`className` A string indicating the name of an ActionScript class.

`constructor` A constructor function used to create an object of a specific class type during object deserialization. The name of the constructor function must be the same as `className`. During object serialization, the name of the constructor function is serialized as the object's type. To unregister the class, pass the value `null` as the `constructor` parameter. Serialization is the process of turning an object into something that you can send to another computer over the network.

Example

The following example defines a `Color` constructor function with properties and methods. After the application connects, the `registerClass()` method is called to register a class for the objects of type `Color`. When a typed object is sent from the client to the server, this class is called to create the server-side object. After the application stops, the `registerClass()` method is called again and passes the value `null` to unregister the class.

```
function Color(){
    this.red = 255;
    this.green = 0;
    this.blue = 0;
}
Color.prototype.getRed = function(){
    return this.red;
}
Color.prototype.getGreen = function(){
    return this.green;
}
Color.prototype.getBlue = function(){
    return this.blue;
}
Color.prototype.setRed = function(value){
    this.red = value;
}
Color.prototype.setGreen = function(value){
    this.green = value;
}
Color.prototype.setBlue = function(value){
    this.blue = value;
}
application.onAppStart = function(){
    application.registerClass("Color", Color);
};
application.onAppStop = function(){
    application.registerClass("Color", null);
};
```

The following example shows how to use the `application.registerClass()` method with the `prototype` property:

```
function A(){}
function B(){}

B.prototype = new A();
// Set constructor back to that of B.
B.prototype.constructor = B;
// Insert code here.
application.registerClass("B", B);
```

application.registerProxy()

```
application.registerProxy(methodName, proxyConnection [, proxyMethodName])
```

Maps a method call to another function. You can use this method to communicate between different application instances that can be on the same Adobe Media Server or on different Adobe Media Servers. Clients can execute server-side methods of any application instances to which they are connected. Server-side scripts can use this method to register methods to be proxied to other application instances on the same server or a different server. You can remove or unregister the proxy by calling this method and passing `null` for the `proxyConnection` parameter, which results in the same behavior as never registering the method at all.

Availability

Flash Communication Server 1

Parameters

`methodName` A string indicating the name of a method. All requests to execute `methodName` for this application instance are forwarded to the `proxyConnection` object.

`proxyConnection` A Client or NetConnection object. All requests to execute the remote method specified by `methodName` are sent to the Client or NetConnection object specified in the `proxyConnection` parameter. Any result returned is sent back to the originator of the call. To unregister or remove the proxy, provide a value of `null` for this parameter.

`proxyMethodName` A string indicating the name of a method for the server to call on the object specified by the `proxyConnection` parameter if `proxyMethodName` is different from the method specified by the `methodName` parameter. This is an optional parameter.

Returns

A value that is sent back to the client that made the call.

Example

In the following example, the `application.registerProxy()` method is called in a function in the `application.onAppStart()` event handler and is executed when the application starts. In the function block, a new NetConnection object called `myProxy` is created and connected. The `application.registerProxy()` method is then called to assign the method `getXyz()` to the `myProxy` object.

```
application.onAppStart = function(){
    var myProxy = new NetConnection();
    myProxy.connect("rtmp://xyz.com/myApp");
    application.registerProxy("getXyz", myProxy);
};
```

application.rejectConnection()

```
application.rejectConnection(clientObj[, description[, errObj])
```

Note: *The* `description` *parameter is supported in Flash Media Server 3 and later.*

Rejects the connection call from a client to the server. The `application.onConnect()` handler is invoked when the client calls `NetConnection.connect()`. In the `application.onConnect()` handler, you can either accept or reject the connection. You can also make a call to an application server to authenticate the client before you accept or reject it.

Note: *When you use version 2 components, the last line (in order of execution) of the* `onConnect()` *handler should be either* `application.acceptConnection()` *or* `application.rejectConnection()` *(unless you're leaving the application in a pending state). Also, any logic that follows* `acceptConnection()` *or* `rejectConnection()` *must be placed in* `application.onConnectAccept()` *and* `application.onConnectReject()` *handlers, or it is ignored. This requirement exists only when you use version 2 components.*

Availability

Flash Communication Server 1

Parameters

`clientObj` A Client object specifying a client to reject.

`description` A string that allows you to provide more information when a connection is redirected.

`errObj` An object of any type that is sent to the client, explaining the reason for rejection. The `errObj` object is available in client-side scripts as the `application` property of the information object that is passed to the `NetConnection.onStatus()` call when the connection is rejected.

Example

In the following example, the client is rejected and sent an error message. This is the server-side code:

```
application.onConnect = function(client){
    // Insert code here.
    var error = new Object();error.message = "Too many connections";
    application.rejectConnection(client, error);
};
```

This is the client-side code:

```
clientConn.onStatus = function (info){
    if (info.code == "NetConnection.Connect.Rejected"){
        trace(info.application.message);
        // Sends the message
        // "Too many connections" to the Output panel
        // on the client side.
    }
};
```

application.sendPeerRedirect()

```
application.sendPeerRedirect(redirectAddress:Array, event:Object)
```

When a peer requests a lookup for a target peer, call this method to send the peer an Array of addresses for the target peer. Call this method from the `application.onPeerLookup()` callback function.

See Distribute peer introductions across multiple servers.

Availability
Flash Media Server 4.5

Parameters
`redirectAddresses` Array; An array of addresses, as Strings, that may be used to contact the target peer.

event Object; The event object received in the `application.onPeerLookup()` callback.

See also
`application.onPeerLookup()`

application.server

`application.server`

Read-only; the platform and version of the server.

Availability
Flash Communication Server 1

Example
The following example checks the `server` property against a string before executing the code in the `if` statement:

```
if (application.server == "Adobe Media Server-Windows/1.0"){
    // Insert code here.
}
```

application.shutdown()

`application.shutdown()`

Unloads the application instance. If the application is running in vhost or application-level scope, only the application instance is unloaded, but the core process remains running. If the application is running in instance scope, the application instance is unloaded and the core process terminates. This process is done asynchronously; the instance is unloaded when the unload sequence begins, not when the `shutdown()` call returns.

After `shutdown()` is called, `application.onAppStop()` is called, connected clients are disconnected, and `application.onDisconnect()` is called for each client. Calls made after calling `shutdown()` may not be executed.

Availability
Flash Media Server 2

Returns
A boolean value indicating success (`true`) or failure (`false`).

ByteArray class

The Server-Side ActionScript ByteArray class is identical to the client-side ByteArray class with the following exceptions:

The following two methods are not implemented in Server-Side ActionScript:

- `ByteArray.inflate()`
- `ByteArray.deflate()`

Where an ActionScript 3.0 ByteArray API uses the int or uint data type, the Server-Side ActionScript ByteArray API uses the Number data type.

To see the methods and properties of the ByteArray class, see the ActionScript 3.0 Reference for the Flash Platform.

Client class

The Client class lets you handle each user, or *client*, connection to an Adobe Media Server application instance. The server automatically creates a Client object when a user connects to an application; the object is destroyed when the user disconnects from the application. Users have unique Client objects for each application to which they are connected. Thousands of Client objects can be active at the same time.

You can use the properties of the Client class to determine the version, platform, and IP address of each client. You can also set individual read and write permissions to various application resources such as Stream objects and shared objects. Use the methods of the Client class to set bandwidth limits and to call methods in client-side scripts.

When you call `NetConnection.call()` from a client-side ActionScript script, the method that is executed in the server-side script must be a method of the Client class. In your server-side script, you must define any method that you want to call from the client-side script. You can also call any methods that you define in the server-side script directly from the Client class instance in the server-side script.

If all instances of the Client class (each client in an application) require the same methods or properties, you can add those methods and properties to the class itself instead of adding them to each instance of a class. This process is called *extending* a class. To extend a class, instead of defining methods in the constructor function of the class or assigning them to individual instances of the class, you assign methods to the `prototype` property of the constructor function of the class. When you assign methods and properties to the `prototype` property, the methods are automatically available to all instances of the class.

The following code shows how to assign methods and properties to an instance of a class. In the `application.onConnect()` handler, the client instance `clientObj` is passed to the server-side script as a parameter. You can then assign a property and method to the client instance.

```
application.onConnect = function(clientObj){
    clientObj.birthday = myBDay;
    clientObj.calculateDaysUntilBirthday = function(){
        // Insert code here.
    }
};
```

The previous example works, but must be executed every time a client connects. If you want the same methods and properties to be available to all clients in the `application.clients` array without defining them every time, assign them to the `prototype` property of the Client class.

There are two steps to extending a built-in class by using the `prototype` property. You can write the steps in any order in your script. The following example extends the built-in Client class, so the first step is to write the function that you will assign to the `prototype` property:

```
// First step: write the functions.

function Client_getWritePermission(){
// The writeAccess property is already built in to the Client class.
    return this.writeAccess;
}

function Client_createUniqueID(){
    var ipStr = this.ip;
// The ip property is already built in to the Client class.
    var uniqueID = "re123mn"
// You would need to write code in the above line
// that creates a unique ID for each client instance.
    return uniqueID;
}

// Second step: assign prototype methods to the functions.

Client.prototype.getWritePermission = Client_getWritePermission;
Client.prototype.createUniqueID = Client_createUniqueID;

// A good naming convention is to start all class method
// names with the name of the class followed by an underscore.
```

You can also add properties to `prototype`, as shown in the following example:

```
Client.prototype.company = "Adobe";
```

The methods are available to any instance, so within `application.onConnect()`, which is passed a `clientObj` parameter, you can write the following code:

```
application.onConnect = function(clientObj){
    var clientID = clientObj.createUniqueID();
    var clientWritePerm = clientObj.getWritePermission();
};
```

Availability

Flash Communication Server 1

Property summary

Property	Description
`Client.agent`	Read-only; the version and platform of the client.
`Client.audioSampleAccess`	Enables Flash Player to access raw, uncompressed audio data from streams in the specified folders.
`Client.farAddress`	Read-only; the derived address from which the server sees the client connection originate.
`Client.farID`	Read-only; a String identifying the RTMFP identity of the server.
`Client.farNonce`	Read-only; a String unique to this client.
`Client.id`	Read-only; a String that uniquely identifies the client.
`Client.ip`	Read-only; a string containing the IP address of the client.
`Client.nearAddress`	Read-only; the public address of the server that the client connected to.
`Client.nearID`	Read-only; a String indicating the RTMFP identity of the server.

Property	Description
`Client.nearNonce`	Read-only.; a String unique to this client.
`Client.pageUrl`	Read-only; a string containing the URL of the web page in which the client SWF file is embedded.
`Client.potentialNearAddresses`	Read-only; the list of all public addresses of the server.
`Client.protocol`	Read-only; a string indicating the protocol used by the client to connect to the server.
`Client.protocolVersion`	Read-only; a string indicating the version of the protocol used by the client to connect to the server.
`Client.readAccess`	A string of directories containing application resources (shared objects and streams) to which the client has read access.
`Client.referrer`	Read-only; a string containing the URL of the SWF file or the server in which this connection originated.
`Client.reportedAddresses`	Read-only; an Array of addresses as Strings of all local addresses at which it can receive RTMFP traffic.
`Client.secure`	Read-only; a boolean value that indicates whether this is an SSL connection (`true`) or not (`false`).
`Client.uri`	Read-only; the URI specified by the client to connect to this application instance.
`Client.videoSampleAccess`	Enables Flash Player to access raw, uncompressed video data from streams in the specified folders.
`Client.virtualKey`	A virtual mapping for clients connecting to the server.
`Client.writeAccess`	Provides write access to directories that contain application resources (such as shared objects and streams) for this client.

Method summary

Method	Description
`Client.call()`	Executes a method on a client or on another server.
`Client.checkBandwidth()`	Call this method from a client-side script to detect bandwidth.
`Client.getBandwidthLimit()`	Returns the maximum bandwidth that the client or the server can use for this connection.
`Client.getStats()`	Returns statistics for the client.
`Client.introducePeer()`	Passes the address for an initiating peer and the tag for its introduction request targeting this client, causing the client to open its end of a P2P connection back to the initiating peer.
`Client.ping()`	Sends a "ping" message to the client and waits for a response.
`Client.remoteMethod()`	Invoked when a client or another server calls the NetConnection.call() method.
`Client.__resolve()`	Provides values for undefined properties.
`Client.setBandwidthLimit()`	Sets the maximum bandwidth for this client from client to server, server to client, or both.

Event summary

Method	Description
`Client.onFarAddressChange()`	Invoked when a client's far address has changed.
`Client.onGroupLeave`	Invoked when a client leaves a NetGroup.
`Client.onGroupJoin`	Invoked with a client joins a NetGroup.
`Client.onReportedAddressChange()`	Invoked when a client reports new addresses.

Client.agent

`clientObject.agent`

Read-only; the version and platform of the client.

When a client connects to the server, the format of `Client.agent` is as follows:

`Operating_System Flash_Player_Version`

For example, if Flash Player version 9.0.45.0 is running on Windows®, the value of `Client.agent` is:

`"WIN 9,0,45,0".`

When a connection is made to another Adobe Media Server, the format of `Client.agent` is as follows:

`Server_Name/Server_Version Operating_System/Operating_System_Build`

For example, if the server version is 3.0.0 and it's running on Windows Server® 2003, the value of `Client.agent` is:

`"FlashCom/3.0.0 WIN/5.1.2600".`

Availability
Flash Communication Server 1

Example
The following example checks the `agent` property against the string `"WIN"` and executes different code depending on whether they match. This code is written in an `onConnect()` function:

```
function onConnect(newClient, name){
    if (newClient.agent.indexOf("WIN") > -1){
        trace ("Window user");
    }   else {
        trace ("non Window user.agent is" + newClient.agent);
    }
}
```

Client.audioSampleAccess

`clientObject.audioSampleAccess`

Enables Flash Player to access raw, uncompressed audio data from streams in the specified folders.

Call the `SoundMixer.computeSpectrum()` method in client-side ActionScript 3.0 to read the raw sound data for a waveform that is currently playing. For more information, see the `SoundMixer.computeSpectrum()` entry in the *ActionScript 3.0 Language and Components Reference* and "Accessing raw sound data" in *Programming ActionScript 3.0*.

Availability
Flash Media Server 3

Example
The following server-side code sets the `audioSampleAccess` directory to `publicdomain`:

```
application.onConnect = function(client) {

        // Anyone can play free content, which is all streams placed under the
        // samples/, publicdomain/ and contrib/ folders.
    client.readAccess = "samples;publicdomain;contrib";

        // Paying customers get to watch more streams.
        if ( isPayingCustomer(client))
        client.readAccess += "nonfree;premium";

        // Content can be saved (user recorded streams) to contrib/ folder.
    client.writeAccess = "contrib";

        // Anyone can gain access to an audio snapshot of the publicdomain/ folder.
    client.audioSampleAccess = "publicdomain";

        // Paying customers can also get a video snapshot of the publicdomain/ folder.
    if (isPayingCustomer(client))
        client.videoSampleAccess = "publicdomain";
}
```

See also

`Client.videoSampleAccess`

Client.call()

```
clientObject.call(methodName, [resultObj, [p1, ..., pN]])
```

Executes a method in client-side code or on another server. The remote method can return data to the `resultObj` parameter, if provided. Whether the remote agent is a client or another server, the method is called on the remote agent's NetConnection object.

In ActionScript 2.0, define the method on the NetConnection object. In ActionScript 3.0, assign the `NetConnection.client` property to an object on which callback methods are invoked. Define the method on that object.

Availability

Flash Communication Server 1

Parameters

`methodName` A string indicating a remote method. The string uses the form `"[objectPath/]method"`. For example, the string `"someObj/doSomething"` tells the client to invoke the `NetConnection.someObj.doSomething()` method on the client or remote server. The string `"doAction"` calls the `doAction()` method on the client or remote server.

`resultObj` An Object. This is an optional parameter that is required when the sender expects a return value from the client. If parameters are passed but no return value is desired, pass the value `null`. The result object can be any object that you define. To be useful, it should have two methods that are invoked when the result arrives: `onResult()` and `onStatus()`. The `resultObj.onResult()` event is triggered if the invocation of the remote method is successful; otherwise, the `resultObj.onStatus()` event is triggered.

`p1, ..., pN` Optional parameters that can be of any ActionScript type, including a reference to another ActionScript object. These parameters are passed to the `methodName` parameter when the method is executed on the Flash client. If you use these optional parameters, you must pass in some value for `resultObj`; if you do not want a return value, pass `null`.

Returns

A boolean value of `true` if a call to `methodName` was successful on the client; otherwise, `false`.

Example

The following ActionScript 2.0 example shows a client-side script that defines a function called `getNumber()` that generates a random number:

```
nc = new NetConnection();
nc.getNumber = function(){
    return (Math.random());
};
nc.connect("rtmp:/clientCall");
```

The following is the same code in ActionScript 3.0:

```
var nc:NetConnection = new NetConnection()
var ncClient = new Object();
nc.client = ncClient;
ncClient.getNumber = nc_getNumber;
function nc_getNumber():void{
    return (Math.random());
}
```

The following server-side script calls `Client.call()` in the `application.onConnect()` handler to call the `getNumber()` method that was defined on the client. The server-side script also defines a function called `randHander()`, which is used in the `Client.call()` method as the `resultObj` parameter.

```
randHandler = function(){
    this.onResult = function(res){
        trace("Random number: " + res);
    }
    this.onStatus = function(info){
        trace("Failed with code:" + info.code);
    }
};
application.onConnect = function(clientObj){
    trace("Connected");
    application.acceptConnection(clientObj);
    clientObj.call("getNumber", new randHandler());
};
```

Note: *This example does not work with version 2 components. For an example of calling `Client.call()` when using version 2 components, see `application.onConnectAccept()`.*

Client.checkBandwidth()

```
clientObject.checkBandwidth()
```

Note: *This method is not supported over RTMFP connections.*

Call this method from a client-side script to detect client bandwidth. If the client is connected directly to the origin server, bandwidth detection occurs on the origin. If the client is connected to the origin server through an edge server, bandwidth detection happens at the first edge to which the client connected.

To use this method to detect client bandwidth, define `onBWDone()` and `onBWCheck()` methods in a client-side script. For more information, see the *Adobe Media Server Developer Guide*.

Note: *If you define the* `checkBandwidth()` *function in a server-side script, the client call runs your definition instead of the definition in the core server code.*

Availability
Flash Media Server 3

Client.farAddress

`clientObject.farAddress`

Read-only. The address from which the server sees the client connection originate. This value is different than `Client.ip` because the value of `Client.farAddress` contains both the IP address and port number for the connection. This property is called the *far address* because from the perspective of the server, the address is on the far side of the NAT or firewall.

Availability
Flash Media Server 4.5

Client.farID

`clientObject.farId`

Read-only. A String identifying the RTMFP identity of the client. This property has the same value as the ActionScript 3.0 `NetConnection.nearID` property. This property is meaningful only for RTMFP connections.

Availability
Flash Media Server 4

Client.farNonce

`clientObject.farNonce`

Read-only. A String unique to this client. This value is defined for RTMFP, RTMPE, and RTMPTE connections.

Availability
Flash Media Server 4

Client.getBandwidthLimit()

`clientObject.getBandwidthLimit(iDirection)`

Note: *This method is not supported over RTMFP connections.*

Returns the maximum bandwidth that the client or the server can use for this connection. Use the `iDirection` parameter to get the value for each direction of the connection. The value returned indicates bytes per second and can be changed with the `Client.setBandwidthLimit()` method. Set the default value for a connection in the Application.xml file of each application.

You can call this method from a client-side script. Call the `NetConnection.call()` method and pass it the name of the method, a result object, and any arguments, as in the following:

```
var re:Responder = new Responder(res);
function res(info) {
     trace(info);
     for (var i:String in info) { trace(i + " - " + info[i]); }
}
nc.call("getBandwidthLimit", re, 0);
```

Availability
Flash Communication Server 1

Parameters
`iDirection` A number specifying the connection direction. The value 0 indicates a client-to-server direction; 1 indicates a server-to-client direction.

Returns
A number.

Example
The following example uses `Client.getBandwidthLimit()` to set the variables `clientToServer` and `serverToClient`:

```
application.onConnect = function(newClient){
    var clientToServer= newClient.getBandwidthLimit(0);var serverToClient=
newClient.getBandwidthLimit(1);
};
```

Client.getStats()

```
clientObject.getStats()
```

Returns statistics for the client.

You can call this method from a client-side script. Call the `NetConnection.call()` method and pass it the name of the method, a result object, and any arguments, as in the following:

```
var re:Responder = new Responder(res);
function res(info) {
     trace(info);
     for (var i:String in info) { trace(i + " - " + info[i]); }
}
nc.call("getStats", re);
```

Availability
Flash Communication Server 1

Returns
An Object with various properties for each statistic returned. The following table describes the properties of the returned object:

Property	Description
bytes_in	Total number of bytes received by this application instance.
bytes_out	Total number of bytes sent from this application instance.
msg_in	Total number of RTMP messages received.
msg_out	Total number of RTMP messages sent.
msg_dropped	Total number of dropped RTMP messages.
ping_rtt	Length of time the client takes to respond to a ping message.
audio_queue_msgs	Current number of audio messages in the queue waiting to be delivered to the client.
video_queue_msgs	Current number of video messages in the queue waiting to be delivered to the client.
so_queue_msgs	Current number of shared object messages in the queue waiting to be delivered to the client.
data_queue_msgs	Current number of data messages in the queue waiting to be delivered to the client.
dropped_audio_msgs	Number of audio messages that were dropped.
dropped_video_msgs	Number of video messages that were dropped.
audio_queue_bytes	Total size of all audio messages (in bytes) in the queue waiting to be delivered to the client.
video_queue_bytes	Total size of all video messages (in bytes) in the queue waiting to be delivered to the client.
so_queue_bytes	Total size of all shared object messages (in bytes) in the queue waiting to be delivered to the client.
data_queue_bytes	Total size of all data messages (in bytes) in the queue waiting to be delivered to the client.
dropped_audio_bytes	Total size of all audio messages (in bytes) that were dropped.
dropped_video_bytes	Total size of all video messages (in bytes) that were dropped.
bw_out	Current downstream bandwidth (outbound from the server).
bw_in	Current upstream bandwidth (inbound to the server) .
client_id	A unique ID issued by the server for this client.

Example

The following example outputs a client's statistics:

```
function testStats(client){
     var stats = client.getStats();
     for(var prop in stats){
          trace("stats." + prop + " = " + stats[prop]);
     }
}
application.onConnect = function(client){
     this.acceptConnection(client);
     testStats(client);
};
```

Client.id

```
clientObject.id
```

Read-only; a string that uniquely identifies the client.

Availability
Flash Media Server 3

Example
The following `onConnect()` function traces the ID of the connecting client:

```
application.onConnect(newClient) {
    trace(newClient.id);
}
```

Client.introducePeer()

```
clientObject.introducePeer(initiatorAddress:String, tag:ByteArray)
```

Opens a peer-to-peer connection with a peer that requested a connection. The peer that requests the connection is called the initiating peer. The initiating peer requests a connection with a target peer. To open the connection, the target peer calls this method and passes the address for the initiating peer and the tag for the introduction request.

Call this method to Distribute peer introductions across multiple servers.

Availability
Flash Media Server 4.5

Parameters
initiator String. The address that the lookup request of the initiating peer came from.

tag ByteArray. The tag identifying the specific lookup request issued by the initiating peer. This value must be handed back in order for the initiating peer to properly correlate and associate the connection attempt from this client to it.

Returns
Nothing.

Client.ip

```
clientObject.ip
```

Read-only; A string containing the IP address of the client.

Availability
Flash Communication Server 1

Example
The following example uses the `Client.ip` property to verify whether a new client has a specific IP address. The result determines which block of code runs.

```
application.onConnect = function(newClient, name){
    if (newClient.ip == "127.0.0.1"){
        // Insert code here.
    } else {
        // Insert code here.
    }
};
```

Client.nearAddress

`clientObject.nearAddress`

Read-only. The public address the client connected to on the server. This address is public, it is not a behind NAT or firewall. This is essential information to generate peer redirects when distributing introductions across multiple servers. The redirect address set must contain known addresses for the target peer as well as the public server address the target peer is connected to. It is called the *near address* because from the perspective of the server it is on the near side of the NAT or firewall.

See Distribute peer introductions across multiple servers.

Availability
Flash Media Server 4.5

Client.nearID

`nc.nearId`

Read-only. A String indicating the RTMFP identity of the server to which the client is connected. This property has the same value as the ActionScript 3.0 `NetConnection.farID` property. This property is meaningful only for RTMFP connections.

Availability
Flash Media Server 4

Client.nearNonce

`nc.nearNonce`

Read-only. A String unique to this client. This value appears to another server as its `Client.farNonce` value. This value is defined for RTMFP, RTMPE, and RTMPTE connections.

Availability
Flash Media Server 4

Client.onFarAddressChange()

```
client.onFarAddressChange = function(){}
```

Invoked when the `farAddress` of a client has changed. For example, a far address changes when a client transitions from a LAN to a wireless connection. RTMFP supports connection mobility so the `farAddress` for a client can change without the connection having to disconnect and reconnect.

Use this event to store a list of client far addresses in a global registry or shared datastore to support distributed peer lookups. See Distribute peer introductions across multiple servers.

Availability
Flash Media Server 4.5

Client.onGroupLeave

```
client.onGroupLeave = function(groupspecDigest:String){}
```

Invoked when a client with an open server channel leaves a group or disconnects.

Availability
Flash Media Server 4.5

Parameters
groupspecDigest A String. The groupspec digest for the group the client is leaving.

Client.onGroupJoin

```
client.onGroupJoin = function(groupcontrol:GroupControl){}
```

Invoked when a client with an open server channel joins a group.

Availability
Flash Media Server 4.5

Parameters
groupcontrol A GroupControl object. The control object representing this Client's membership within a group..

Example

```
var groups = {};

Client.prototype.onGroupJoin = function(groupControl)
{
    groupControl["client"] = this; // Remember the associated Client.
    var groupControlArray = groups[groupControl.groupspecDigest];
    if (groupControlArray)
    {
        trace("Register Client in existing Group (by groupspec digest): " +
              groupControl.groupspecDigest +
              ", current Group size is: " +
              groupControlArray.length);

        // find a random member to bootstrap with
        r = Math.random();
        index = Math.floor(r * groupControlArray.length);

        var peerGroupControl = groupControlArray[index];
        groupControl.addNeighbor(peerGroupControl["client"].farID);
        groupControlArray.push(groupControl);
    }
    else
    {
        trace("Track client joining new Group (by groupspec digest): " +
groupControl.groupspecDigest);

        groupControlArray = [];
        groupControlArray.push(groupControl);
        groups[groupControl.groupspecDigest] = groupControlArray;
    }
}
```

Client.onReportedAddressChange()

```
client.onReportedAddressChange = function(){}
```

Invoked when a client reports new addresses.

Use this event to store a list of client addresses in a global registry or shared datastore to support distributed peer lookups. See Distribute peer introductions across multiple servers.

Availability

Flash Media Server 4.5

Example

```
client.onReportedAddressesChange = function() {
    var newReportedAddresses = this.reportedAddresses;
   // Now store these in a global registry or shared datastore to support distributed scripted
peer lookups.
    // ...
}
```

Client.pageUrl

```
clientObject.pageUrl
```

Read-only; A string containing the URL of the web page in which the client SWF file is embedded. If the SWF file isn't embedded in a web page, the value is the location of the SWF file. The following code shows the two examples:

```
// trace.swf file is embedded in trace.html.
client.pageUrl: http://www.example.com/trace.html

// trace.swf is not embedded in an html file.
client.pageUrl: http://www.example.com/trace.swf
```

The value cannot be a local file address.

Availability

Flash Media Server 2

Example

The following example uses the `Client.pageUrl` property to verify whether a new client is located at a particular URL. The result determines which block of code runs.

```
application.onConnect = function(newClient){
    if (newClient.pageUrl == "http://www.example.com/index.html"){
        return true;
    } else {
        return false;
    }
};
```

Client.ping()

```
clientObject.ping()
```

Sends a "ping" message to the client and waits for a response. If the client responds, the method returns `true`; otherwise, `false`. Use this method to determine whether the client connection is still active.

Availability

Flash Communication Server 1

Example

The following `onConnect()` function pings the connecting client and traces the results of the method:

```
application.onConnect(newClient) {
    if (newClient.ping()){
        trace("ping successful");
    }
    else {
        trace("ping failed");
    }
}
```

See also

`Client.getStats()`

Client.potentialNearAddresses

`clientObject.potentialNearAddress`

Read-only; the list of all public addresses of the server. The `nearAddress` is the public address of the interface to which the client is connected. However the `potentialNearAdresses` is the list of all the public interfaces that may be used to communicate with the server this client is connected to.

Use this property to distribute peer introductions across multiple servers. See Distribute peer introductions across multiple servers.

Availability

Flash Media Server 4.5

Example

The following example outputs all the potential near addresses of the server the client has connected to:

```
function logAllPotentialNearAddresses(client) {
    var n = client.potentialNearAddresses.length;
   trace("Client has " + n + " potential near addresses (at the server-end of its connection).");
    for (var i = 0; i < n; ++i)
        trace("  " + i + " : " + client.potentialNearAddresses[i] + "\n ");
}
```

Client.protocol

`clientObject.protocol`

Read-only; A string indicating the protocol used by the client to connect to the server. This string can have one of the following values:

Protocol	Description
`rtmp`	RTMP over a persistent socket connection.
`rtmpt`	RTMP tunneled over HTTP.
`rtmps`	RTMP over an SSL (Secure Socket Layer) connection.
`rtmpe`	An encrypted RTMP connection.
`rtmpte`	An encrypted RTMP connection tunneled over HTTP.
rtmfp	Real-Time Media Flow Protocol.

Availability
Flash Communication Server 1

Example
The following example checks the connection protocol used by a client upon connection to the application:

```
application.onConnect(clientObj){
    if(clientObj.protocol == "rtmp") {
        trace("Client connected over RTMP");
    } else if(clientOjb.protocol == "rtmpt") {
        trace("Client connected over RTMP tunneled over HTTP");
    }
}
```

Client.protocolVersion

`clientObject.protocolVersion`

Read-only; A string indicating the version of the protocol used by the client to connect to the server. This value matches the value in the `c-proto-ver` field in the Access log.

See Fields in access logs.

Availability
Flash Media Server 4

Client.readAccess

`clientObject.readAccess`

Gives clients read access to directories containing shared objects and streams. You cannot specify file names, you can specify only a directory or a path to a directory (for example, "directory" or "directory/subdir/subdir2"). The directory you specify grants read access to that directory and to all its subdirectories. To give a client read access to multiple directories, list the directories in a string delimited by semicolons.

The default value is `"/"`. This value grants read access to the directories in which the server is configured to look for streams and shared objects.

Note: *Adobe recommends that you store either streams or shared objects in a directory, but not both.*

A directory you specify is relative to the directory in which the server is configured to store streams or shared objects for that application instance. If you use a virtual directory or a storage directory, the `readAccess` value is relative to that location.

By default, the server stores persistent shared objects in the *rootinstalldir*\applications*appname*\sharedobjects_definst_ directory.

By default, the server looks for streams for the default application instance in the directory *rootinstalldir*\applications*appname*\streams_definst_. For example, a client that connects to "rtmp://someamssever.com/test" looks for streams in the *rootinstalldir*\applications\test\streams_definst_ directory. A client that connects to "rtmp://someamsserver.com/test/room1" looks for streams in the *rootinstalldir*\applications\test\streams\room1 directory.

Suppose there is a stream called "sample.f4v" in the applications\test\streams_definst_ directory. In the server-side script, if you give `client.readAccess` any value other than "/", the stream does not play.

Note: *If you specify `"\"`, the script does not run.*

Suppose you copy the file sample2.f4v into the directory test/streams/_definst_/protected. In the server-side script, set `client.readAccess="protected"`. In the client-side script, call `netstream.play("mp4:protected/sample2.f4v")`. The file plays because it's located in a directory that has read access.

Now call `netstream.play("mp4:sample.f4v")`. The file does not play because the test/streams/_definst_ directory does not have read access.

Availability
Flash Communication Server 1

Details
To give a client read access, specify a list of directories (in URI format), delimited by semicolons. Any files or directories within a specified URI are also considered accessible. For example, if you specify `"myMedia"`, any files or directories in the myMedia directory are also accessible (for example, myMedia/mp3s). Any files or directories in the myMedia/mp3s directory are also accessible, and so on.

Clients with read access to a directory that contains streams can play the streams. Clients with read access to a directory that contains shared objects can subscribe to the shared objects and receive notification of changes in the shared objects.

- For streams, `readAccess` controls the streams that the connection can play.
- For shared objects, `readAccess` controls whether the connection can listen to shared object changes.

To control access for a particular file, create a separate directory for the file and set `readAccess` to that directory.

Note: *You cannot set this property in the `application.onPublish()` event.*

Example
The following code is for an application called "amsapp". It gives clients read access to all files in the folders mymedia/mp3s and mydata/notes. The clients also have read access to any files in subfolders of those folders.

```
application.onConnect = function(newClient, name){
    newClient.readAccess = "mymedia/mp3s;mydata/notes";
};
```

Clients that connect to an instance of the application "amsapp" can play streams in the folder *rootinstall*/applications/amsapp/streams/*instancename*/mymedia/mp3s and all its subfolders. Those clients can listen for changes to shared objects in the folder *rootinstall*/applications/amsapp/sharedobjects/*instancename*/mydata/notes and all its subfolders.

Client.referrer

`clientObject.referrer`

Read-only; A string containing the URL of the SWF file or the server in which this connection originated. The property is set when a SWF hosted on a web server or connects to an application on Adobe Media Server. The property is also set when one Adobe Media Server instance connects to another.

This property is not set when a SWF from a local file system running in stand-alone Flash Player version 10 or above connects to Adobe Media Server. If a SWF file is running in standalone Flash Player version 8 or 9, the property is set as `file:///...`.

Availability

Flash Communication Server 1

Example

```
application.onConnect = function(newClient, name){
    trace("New user connected to server from" + newClient.referrer);
};
```

Client.remoteMethod()

`myClient.remoteMethod = function([p1, ..., pN]){}`

You can define methods on the Client object and call the methods from client-side code. To call methods from client-side code, call the `NetConnection.call()` method and pass it the name of the method you defined. The server searches the Client object instance for the method. If the method is found, it is invoked and the return value is sent back to the result object specified in the call to `NetConnection.call()`.

Availability

Flash Communication Server 1

Parameters

`p1, ..., pN` Optional parameters passed to the `NetConnection.call()` method.

Example

The following example creates a method called `sum()` as a property of the Client object `newClient` on the server side:

```
Client.prototype.sum = function(op1, op2){
    return op1 + op2;
};
```

You can call the server-side `sum()` method from a client-side call to the `NetConnection.call()` method:

```
nc = new NetConnection();
nc.connect("rtmp://myServer/myApp");
nc.call("sum", new result(), 20, 50);
function result(){
    this.onResult = function (retVal){
        output += "sum is " + retVal;
    };
    this.onStatus = function(errorVal){
        output += errorVal.code + " error occurred";
    };
}
```

You can also call the `sum()` method in server-side code:

```
newClient.sum();
```

The following example creates two functions that you can call from either a client-side or server-side script:

```
application.onConnect = function(clientObj) {
    // The function foo returns 8.
    clientObj.foo = function() {return 8;};
    // The function bar is defined outside the onConnect call.
    clientObj.bar = application.barFunction;
};
// The bar function adds the two values it is given.
application.barFunction = function(v1,v2) {
    return (v1 + v2);
};
```

You can call either of the two functions that were defined in the previous example (`foo` and `bar`) by using the following code in a client-side script:

```
c = new NetConnection();
c.call("foo");
c.call("bar", null, 1, 1);
```

You can call either of the two functions that were defined in the previous example (`foo` and `bar`) by using the following code in a server-side script:

```
c = new NetConnection();
c.onStatus = function(info) {
    if(info.code == "NetConnection.Connect.Success") {
        c.call("foo");
        c.call("bar", null, 2, 2);
    }
};
```

Client.reportedAddresses

```
clientObject.reportedAddresses
```

Read-only; an Array of Strings containing all the addresses at which a client can receive RTMFP traffic. The client can update this value multiple times over the lifetime of its RTMFP connection to the server.

There is a small time lag between when the client is connected and when it reports its IP addresses. The time lag is usually a few hundred milliseconds. When the server receives the reported addresses from the client, it gets an `Client.onReportedAddressChange()` event. The reported addresses are valid only after the first `onReportedAddressChange()` event.

Use this property to distribute peer introductions across multiple servers. See Distribute peer introductions across multiple servers.

Availability

Flash Media Server 4.5

Example

The following function outputs a list of all the reported addresses for a client:

```
function logReportedAddresses(client) {
    var n = client.reportedAddresses.length;
    trace("Client has reported " + n + " addresses.");
    for (var i = 0; i < n; ++i)
        trace("  " + i + ": " + client.reportedAddresses[i]);
}
```

Client.__resolve()

```
Client.__resolve = function(propName){}
```

Provides values for undefined properties. When an undefined property of a Client object is referenced by Server-Side ActionScript code, the Client object is checked for a `_resolve()` method. If the object has a `_resolve()` method, it is invoked and passed the name of the undefined property. The return value of the `_resolve()` method is the value of the undefined property. In this way, `_resolve()` can supply the values for undefined properties and make it appear as if they are defined.

Availability
Flash Communication Server 1

Parameters
`propName` A string indicating the name of an undefined property.

Returns
The value of the property specified by the `propName` parameter.

Example
The following example defines a function that is called whenever an undefined property is referenced:

```
Client.prototype.__resolve = function (name) {
    return "Hello, world!";
};
function onConnect(newClient){
// Prints "Hello World".
    trace (newClient.property1);
}
```

Client.secure

```
clientObject.secure
```

Read-only; A boolean value that indicates whether this is an SSL connection (`true`) or not (`false`).

Availability
Flash Media Server 2

Client.setBandwidthLimit()

```
clientObject.setBandwidthLimit(iServerToClient, iClientToServer)
```

Note: *This method is not supported over RTMFP connections.*

Sets the maximum bandwidth for this client from client to server, server to client, or both. The default value for a connection is set for each application in the `Client` section of the Application.xml file. The value specified cannot exceed the bandwidth cap value specified in the Application.xml file. For more information, see `BandwidthCap` in the *Adobe Media Server Configuration and Administration Guide*.

You can call this method from a client-side script. Call the `NetConnection.call()` method and pass it the name of the method, a result object, and any arguments, as in the following:

```
var re:Responder = new Responder(res);
function res(info) {
     trace(info);
     for (var i:String in info) { trace(i + " - " + info[i]); }
}
nc.call("setBandwidthLimit", re, 125000, 125000);
```

Availability
Flash Communication Server 1

Parameters
`iServerToClient` A number; the bandwidth from server to client, in bytes per second. Use 0 if you don't want to change the current setting.

`iClientToServer` A number; the bandwidth from client to server, in bytes per second. Use 0 if you don't want to change the current setting.

Example
The following example sets the bandwidth limits for each direction, based on values passed to the `onConnect()` function:

```
application.onConnect = function(newClient, serverToClient, clientToServer){
    newClient.setBandwidthLimit(serverToClient, clientToServer);
    application.acceptConnection(newClient);
}
```

Client.uri

`clientObject.uri`

Read-only; the URI specified by the client to connect to this application instance.

Availability
Flash Media Server 2

Example
The following example defines an `onConnect()` callback function that sends a message indicating the URI that the new client used to connect to the application:

```
application.onConnect = function(newClient, name){
    trace("New user requested to connect to " + newClient.uri);
};
```

Client.videoSampleAccess

`clientObject.videoSampleAccess`

Enables Flash Player to access raw, uncompressed video data from streams in the specified folders.

Call the `BitmapData.draw()` method in client-side ActionScript 3.0 to read the raw data for a stream that is currently playing. For more information, see the `BitmapData.draw()` entry in *ActionScript 3.0 Language and Components Reference*.

Availability
Flash Media Server 3

Example
The following server-side code sets the `videoSampleAccess` directory to `publicdomain` for paying customers:

```
application.onConnect = function(client) {

        // Anyone can play free content, which is all streams placed under the
        // samples/, publicdomain/, and contrib/ folders.
    client.readAccess = "samples;publicdomain;contrib";

        // Paying customers get to watch more streams.
        if ( isPayingCustomer(client))
        client.readAccess += "nonfree;premium";

        // Content can be saved (user recorded streams) to the contrib/ folder.
    client.writeAccess = "contrib";

        // Anyone can gain access to an audio snapshot of the publicdomain/ folder.
    client.audioSampleAccess = "publicdomain";

        // Paying customers can also get a video snapshot of the publicdomain/ folder.
    if (isPayingCustomer(client))
        client.videoSampleAccess = "publicdomain";
}
```

See also
`Client.audioSampleAccess`

Client.virtualKey

`clientObject.virtualKey`

Use this property in conjunction with the `Stream.setVirtualPath()` method to map stream URLs to physical locations on the server. This allows you to serve different content to different versions of Flash Player.

When a client connects, it receives a virtual key that corresponds to ranges that you set in the Vhost.xml file. You can use `Client.virtualKey` to change that value in a server-side script. The following is the code in the Vhost.xml file that you must configure:

```
<VirtualKeys>
    <!-- Create your own ranges and key values.-->
    <!-- You can create as many Key elements as you need.-->
    <Key from="WIN 7,0,19,0" to="WIN 9,0,0,0">A</Key>
</VirtualKeys>
```

Using the previous Vhost.xml file, if a Flash Player 8 client connected to the server, its `Client.virtualKey` value would be `A`.

Note: *A legal key cannot contain the characters "*" and ":".*

Availability
Flash Media Server 2

Client.writeAccess

`clientObject.writeAccess`

Provides write access to directories that contain application resources (such as shared objects and streams) for this client. To give a client write access to directories that contain application resources, list directories in a string delimited by semicolons. By default, all clients have full write access, and the `writeAccess` property is set to slash (`/`). For example, if `myMedia` is specified as an access level, then any files or directories in the myMedia directory are also accessible (for example, myMedia/myStreams). Similarly, any files or subdirectories in the myMedia/myStreams directory are also accessible, and so on.

- For shared objects, `writeAccess` provides control over who can create and update the shared objects.
- For streams, `writeAccess` provides control over who can publish and record a stream.

You cannot use this property to control access to a single file. To control access to a single file, create a separate directory for the file.

Don't precede the stream path with a leading slash (/) on the client side.

Note: *You cannot set this property in the `application.onPublish()` event.*

Availability
Flash Communication Server 1

Example
The following example provides write access to the /myMedia/myStreams and myData/notes directories:

```
application.onConnect = function(newClient, name){
    newClient.writeAccess = "/myMedia/myStreams;myData/notes";
    application.acceptConnection();
};
```

The following example completely disables write access:

```
application.onConnect = function(clientObj){
    clientObj.writeAccess = "";
    return true;
};
```

See also
`Client.readAccess`

File class

The File class lets applications write to the server's file system. This is useful for storing information without using a database server, creating log files for debugging, and tracking usage. Also, a directory listing is useful for building a content list of streams or shared objects without using Flash Remoting.

By default, a script can access files and directories only within the application directory of the hosting application. A server administrator can grant access to additional directories by specifying virtual directory mappings for File object paths. This is done in the `FileObject` tag in the Application.xml file, as shown in the following example:

```
<FileObject>
    <VirtualDirectory>/videos;C:\myvideos</VirtualDirectory>
    <VirtualDirectory>/amsapps;C:\Program Files\ams\applications</VirtualDirectory>
</FileObject>
```

This example specifies two additional directory mappings in addition to the default application directory. Any path that begins with /videos—for example, /videos/xyz/vacation.flv—maps to c:/myvideos/xyz/vaction.flv. Similarly, /amsapps/conference maps to c:/Program Files/ams/applications/conference. Any path that does not match a mapping resolves to the default application folder. For example, if c:/myapps/filetest is the application directory, then /streams/hello.flv maps to c:/myapps/filetest/streams/hello.flv.

Note: *You can use an Application.xml file at the virtual host level or at the application level.*

In addition, the following rules are enforced by the server:

- File objects cannot be created by using native file path specification.
- File object paths must follow the URI convention:

  A slash (/) must be used as the path separator. Access is denied if a path contains a backslash (\), or if a dot (.) or two dots (..) is the only string component found between path separators.
- Root objects cannot be renamed or deleted.

  For example, if a path using a slash (/) is used to create a File object, the application folder is mapped.

Availability

Flash Media Server 2

Property summary

Property	Description
`File.canAppend`	Read-only; a boolean value indicating whether a file can be appended (`true`) or not (`false`).
`File.canRead`	Read-only; A boolean value indicating whether a file can be read (`true`) or not (`false`).
`File.canReplace`	Read-only; A boolean value indicating whether a file was opened in `"create"` mode (`true`) or not (`false`). This property is undefined for closed files.
`File.canWrite`	Read-only; a boolean value indicating whether a file can be written to (`true`) or not (`false`).
`File.creationTime`	Read-only; a Date object containing the time the file was created.
`File.exists`	Read-only; a boolean value indicating whether the file or directory exists (`true`) or not (`false`).
`File.isDirectory`	Read-only; a boolean value indicating whether the file is a directory (`true`) or not (`false`).
`File.isFile`	Read-only; a boolean value indicating whether the file is a regular data file (`true`) or not (`false`).
`File.isOpen`	Read-only; a boolean value indicating whether the file has been successfully opened and is still open (`true`) or not (`false`).
`File.lastModified`	Read-only; a Date object containing the time the file was last modified.
`File.length`	Read-only; for a directory, the number of files in the directory, not counting the current directory and parent directory entries; for a file, the number of bytes in the file.
`File.mode`	Read-only; the mode of an open file.

Property	Description
`File.name`	Read-only; a string indicating the name of the file.
`File.position`	The current offset in the file.
`File.type`	Read-only; a string specifying the type of data or encoding used when a file is opened.

Method summary

Method	Description
`File.close()`	Closes the file.
`File.copyTo()`	Copies a file to a different location or copies it to the same location with a different filename.
`File.eof()`	Returns a boolean value indicating whether the file pointer is at the end of file (`true`) or not (`false`).
`File.flush()`	Flushes the output buffers of a file.
`File.list()`	If the file is a directory, lists the files in the directory.
`File.mkdir()`	Creates a directory.
`File.open()`	Opens a file so that you can read from it or write to it.
`File.read()`	Reads the specified number of characters from a file and returns a string.
`File.readAll()`	Reads the file after the location of the file pointer and returns an array with an element for each line of the file.
`File.readByte()`	Reads the next byte from the file and returns the numeric value of the next byte, or -1 if the operation fails.
`File.readBytes()`	Reads a specified number of bytes from a file into a ByteArray.
`File.readln()`	Reads the next line from the file and returns it as a string.
`File.remove()`	Removes the file or directory pointed to by the File object.
`File.renameTo()`	Moves or renames a file.
`File.seek()`	Skips a specified number of bytes and returns the new file position.
`File.toString()`	Returns the path to the File object.
`File.write()`	Writes data to a file.
`File.writeAll()`	Takes an array as a parameter and calls the `File.writeln()` method on each element in the array.
`File.writeByte()`	Writes a byte to a file.
`File.writeBytes()`	Writes a specified number of bytes to a file from a ByteArray.
`File.writeln()`	Writes data to a file and adds a platform-dependent end-of-line character after outputting the last parameter.

File constructor

fileObject = new File(`name`)

Creates an instance of the File class.

Availability

Flash Media Server 2

Parameters

`name` A string indicating the name of the file or directory. The name can contain only UTF-8 encoded characters; high byte values can be encoded by using the URI character-encoding scheme. The specified name is mapped to a system path by using the mappings specified in the `FileObject` section of the Application.xml file. If the path is invalid, the `name` property of the object is set to an empty string, and no file operation can be performed.

Returns

A File object if successful; otherwise, `null`.

Example

The following code creates an instance of the File class:

```
var errorLog = new File("/logs/error.txt");
```

Note that the physical file isn't created on the hard disk until you call `File.open()`.

File.canAppend

`fileObject.canAppend`

Read only; a boolean value indicating whether a file can be appended (`true`) or not (`false`). The property is undefined for closed files.

Availability

Flash Media Server 2.0

File.canRead

`fileObject.canRead`

Read-only; A boolean value indicating whether a file can be read (`true`) or not (`false`).

Availability

Flash Media Server 2

File.canReplace

`fileObject.canReplace`

Read-only; A boolean value indicating whether a file was opened in `"create"` mode (`true`) or not (`false`). This property is undefined for closed files.

Availability

Flash Media Server 2

File.canWrite

`fileObject.canWrite`

Read only; a boolean value indicating whether a file can be written to (`true`) or not (`false`).

Note: *If `File.open()` was called to open the file, the mode in which the file was opened is respected. For example, if the file was opened in read mode, you can read from the file, but you cannot write to the file.*

Availability
Flash Media Server 2

File.close()

```
fileObject.close()
```

Closes the file. This method is called automatically on an open File object when the object is out of scope.

Availability
Flash Media Server 2

Returns
A boolean value indicating whether the file was closed successfully (`true`) or not (`false`). Returns `false` if the file is not open.

Example
The following code closes the /path/file.txt file:

```
if (x.open("/path/file.txt", "read") ){
    // Do something here.
    x.close();
}
```

File.copyTo()

```
fileObject.copyTo(name)
```

Copies a file to a different location or copies it to the same location with a different filename. This method returns `false` if the source file doesn't exist or if the source file is a directory. When this method fails, it invokes the `application.onStatus()` event handler to report errors.

Note: *The user or process owner that the server runs under in the operating system must have adequate write permissions or the call can fail.*

Availability
Flash Media Server 2

Parameters
`name` Specifies the name of the destination file. The name can contain only UTF-8 characters; high byte values can be encoded by using the URI character-encoding scheme. The name specified is mapped to a system path by using the mappings specified in the Application.xml file. If the path is invalid or if the destination file doesn't exist, the operation fails, and the method returns `false`.

Returns
A boolean value indicating whether the file is copied successfully (`true`) or not (`false`).

Example
The following code copies the file set by the `myFileObj` File object to the location provided by the parameter:

```
if (myFileObj.copyTo( "/logs/backup/hello.log")){
    // Do something here.
}
```

File.creationTime

`fileObject.creationTime`

Read-only; a Date object containing the time the file was created.

Availability
Flash Media Server 2

File.eof()

`fileObject.eof()`

Returns a boolean value indicating whether the file pointer is at the end of file (`true`) or not (`false`). If the file is closed, the method returns `true`.

Availability
Flash Media Server 2

Returns
A boolean value.

Example
The following `while` statement lets you insert code that is executed until the file pointer is at the end of a file:

```
while (!myFileObj.eof()){
// Do something here.
}
```

File.exists

`fileObject.exists`

Read-only; a boolean value indicating whether the file or directory exists (`true`) or not (`false`).

Availability
Flash Media Server 2

File.flush()

`fileObject.flush()`

Flushes the output buffers of a file. If the file is closed, the operation fails. When this method fails, it invokes the `application.onStatus()` event handler to report errors.

Availability
Flash Media Server 2

Returns
A boolean value indicating whether the flush operation was successful (`true`) or not (`false`).

File.isDirectory

`fileObject.isDirectory`

Read-only; a boolean value indicating whether the file is a directory (`true`) or not (`false`).

A File object that represents a directory has properties that represent the files contained in the directory. These properties have the same names as the files in the directory, as shown in the following example:

```
myDir = new File("/some/directory");
myFileInDir = myDir.fileName;
trace(myDir.isDirectory) // Outputs true.
```

The following example uses named property lookup to refer to files that do not have valid property names:

```
mySameFileInDir = myDir["fileName"];
myOtherFile = myDir["some long filename with spaces"];
```

Availability
Flash Media Server 2

File.isFile

`fileObject.isFile`

Read-only; a boolean value indicating whether a file is a regular data file (`true`) or not (`false`).

Availability
Flash Media Server 2

File.isOpen

`fileObject.isOpen`

Read-only; a boolean value indicating whether the file has been successfully opened and is still open (`true`) or not (`false`).

Note: *Directories do not need to be opened.*

Availability
Flash Media Server 2

File.lastModified

`fileObject.lastModified`

Read-only; a Date object containing the time the file was last modified.

Availability
Flash Media Server 2

File.length

```
fileObject.length
```

Read-only; for a directory, the number of files in the directory, not counting the current directory and parent directory entries; for a file, the number of bytes in the file.

Availability
Flash Media Server 2

File.list()

```
fileObject.list(filter)
```

If the file is a directory, lists the files in the directory. Returns an array with an element for each file in the directory.

Availability
Flash Media Server 2

Parameters
`filter` A Function object that determines the files in the returned array.

If the function returns `true` when a file's name is passed to it as a parameter, the file is added to the array returned by `File.list()`. This parameter is optional and allows you to filter the results of the call.

Returns
An Array object.

Example
The following example returns files in the current directory that have 3-character names:

```
var a = x.currentDir.list(function(name){return name.length==3;});
```

File.mkdir()

```
fileObject.mkdir(newDir)
```

Creates a directory. The directory is created in the directory specified by `fileObject`. When this method fails, it invokes the `application.onStatus()` event handler to report errors.

The user or process owner that the server runs under in the operating system must have adequate write permissions or the call can fail.

Note: *You cannot call this method from a File object that is a file (where `isFile` is `true`). You must call this method from a File object that is a directory (where `isDirectory` is `true`).*

Availability
Flash Media Server 2

Parameters
`newDir` A string indicating the name of the new directory. This name is relative to the current File object instance.

Returns
A boolean value indicating success (`true`) or failure (`false`).

Example
The following example creates a logs directory in the `myFileObject` instance:

```
if (myFileObject.mkdir("logs")){
    // Do something if a logs directory is created successfully.
}
```

File.mode

`fileObject.mode`

Read-only; the mode of an open file. It can be different from the `mode` parameter that was passed to the `open()` method for the file if you have repeating attributes (for example, `"read, read"`) or if some attributes were ignored. If the file is closed, the property is `undefined`.

Availability
Flash Media Server 2

See also
`File.open()`

File.name

`fileObject.name`

Read-only; a string indicating the name of the file. If the File object was created with an invalid path, the value is an empty string.

Availability
Flash Media Server 2

File.open()

`fileObject.open(type, mode)`

Opens a file so that you can read from it or write to it. First use the File constructor to create a File object and then call `open()` on that object. When the `open()` method fails, it invokes the application.onStatus() event handler to report errors.

Availability
Flash Media Server 2

Parameters
`type` A string indicating the encoding type for the file. The following types are supported (there is no default value):

Value	Description
`"text"`	Opens the file for text access by using the default file encoding.
`"binary"`	Opens the file for binary access.
`"utf8"`	Opens the file for UTF-8 access.

`mode` A string indicating the mode in which to open the file. The following modes are valid and can be combined (modes are case sensitive and multiple modes must be separated by commas—for example, `"read,write"`; there is no default value):

Value	Description
`"read"`	Opens a file for reading.
`"write"`	Opens a file for writing.
`"readWrite"`	Opens a file for both reading and writing.
`"append"`	Opens a file for writing and positions the file pointer at the end of the file when you attempt to write to the file.
`"create"`	Creates a new file if the file is not present. If a file exists, its contents are destroyed and a new file is created.

Note: *If both `"read"` and `"write"` are set, `"readWrite"` is automatically set. The user or process owner that the server runs under in the operating system must have write permissions to use `"create"`, "append", "readWrite", and "write" modes.*

Returns

A boolean value indicating whether the file opened successfully (`true`) or not (`false`).

Example

The following client-side script creates a connection to an application called file:

```
var nc:NetConnection = new NetConnection();
function traceStatus(info) {
    trace("Level: " + info.level + " Code: " + info.code);
}
nc.onStatus = traceStatus;
nc.connect("rtmp:/file");
```

The following server-side script creates a text file called log.txt and writes text to the file:

```
application.onConnect = function(client){
    this.acceptConnection(client);
    var logFile = new File("log.txt");
    if(!logFile.exists){
        logFile.open("text", "append");
        logFile.write("something", "somethingElse")
    }
};
```

File.position

`fileObject.position`

The current offset in the file. This is the only property of the File class that can be set. Setting this property performs a seek operation on the file. The property is undefined for closed files.

Availability
Flash Media Server 2

File.read()

```
fileObject.read(numChars)
```

Reads the specified number of characters from a file and returns a string. If the file is opened in binary mode, the operation fails. When this method fails, it invokes the `application.onStatus()` event handler to report errors.

Availability
Flash Media Server 2

Parameters
`numChars` A number specifying the number of characters to read. If `numChars` specifies more bytes than are left in the file, the method reads to the end of the file.

Returns
A string.

Example
The following code opens a text file in read mode and sets variables for the first 100 characters, a line, and a byte:

```
if (myFileObject.open("text", "read")){
    strVal = myFileObject.read(100);
    strLine = myFileObject.readln();
    strChar = myFileObject.readByte();
}
```

File.readAll()

```
fileObject.readAll()
```

Reads the file after the location of the file pointer and returns an Array object with an element for each line of the file. If the file opened in binary mode, the operation fails. When this method fails, it invokes the `application.onStatus()` event handler to report errors.

Availability
Flash Media Server 2

Returns
An Array object.

File.readByte()

```
fileObject.readByte()
```

Reads the next byte from the file and returns the numeric value of the next byte or -1if the operation fails. If the file is not opened in binary mode, the operation fails.

Availability
Flash Media Server 2

Returns
A number; either a positive integer or -1.

File.readBytes()

```
fileObject.readBytes(dest, offset, length)
```

Reads the number of bytes specified by the `length` parameter from the `fileObject` into the `dest` parameter starting at the `offset` within `dest`. This method throws an EOFError if `length` exceeds `File.length - File.position`.

Availability
Flash Media Server 4

Parameters
`dest` A ByteArray into which bytes from the `fileObject` are read.

`offset` An integer specifying an offset location within the ByteArray specified in the `dest` parameter. This parameter is optional. The default value is 0.

`length` An integer specifying the number of bytes to read from the `fileObject`. This parameter is optional. The default value is `File.length - File.position`.

Returns
Nothing.

File.readln()

```
fileObject.readln()
```

Reads the next line from the file and returns it as a string. The line-separator characters (either \r\n on Windows or \n on Linux) are not included in the string. The character \r is skipped; \n determines the end of the line. If the file opened in binary mode, the operation fails.

The `File.readln()` method has a maximum character limit of around 4100 characters. To read more characters, call `File.readAll().join('')`.

Availability
Flash Media Server 2

Returns
A string.

File.remove()

```
fileObject.remove(recursive)
```

Removes the file or directory pointed to by the File object. When this method fails, it invokes the `application.onStatus()` event handler to report errors.

Availability
Flash Media Server 2

Parameters

`recursive` A boolean value specifying whether to do a recursive removal of the directory and all its contents (`true`), or a nonrecursive removal of the directory contents (`false`). If no value is specified, the default value is `false`. If `fileObject` is not a directory, any parameters passed to the `remove()` method are ignored.

Returns

A boolean value indicating whether the file or directory was removed successfully (`true`) or not (`false`). Returns `false` if the file is open, the path points to a root folder, or the directory is not empty.

Example

The following example shows the creation and removal of a file:

```
fileObject = new File("sharedobjects/_definst_/userIDs.fso");
fileObject.remove();
```

File.renameTo()

```
fileObject.renameTo(name)
```

Moves or renames a file. If the file is open or the directory points to the root directory, the operation fails. When this method fails, it invokes the `application.onStatus()` event handler to report errors.

Availability

Flash Media Server 2

Parameters

`name` The new name for the file or directory. The name can contain only UTF-8-encoded characters; high byte values can be encoded by using the URI character-encoding scheme. The specified name is mapped to a system path by using the mappings specified in the Application.xml file. If the path is invalid or the destination file doesn't exist, the operation fails.

Returns

A boolean value indicating whether the file was successfully renamed or moved (`true`) or not (`false`).

File.seek()

```
fileObject.seek(numBytes)
```

Skips a specified number of bytes and returns the new file position. This method can accept both positive and negative parameters.

Availability

Flash Media Server 2

Parameters

`numBytes` A number indicating the number of bytes to move the file pointer from the current position.

Returns
If the operation is successful, returns the current position in the file; otherwise, returns -1. If the file is closed, the operation fails and calls `application.onStatus()` to report a warning. The operation returns -1 when called on a directory.

File.toString()

```
fileObject.toString()
```

Returns the path to the File object.

Availability
Flash Media Server 2

Returns
A string.

Example
The following example outputs the path of the File object `myFileObject`:

```
trace(myFileObject.toString());
```

File.type

```
fileObject.type
```

Read-only; a string specifying the type of data or encoding used when a file is opened. The following strings are supported: `"text"`, `"utf8"`, and `"binary"`. This property is undefined for directories and closed files. If the file is opened in `"text"` mode and UTF-8 BOM (Byte Order Mark) is detected, the `type` property is set to `"utf8"`.

Availability
Flash Media Server 2.0

See also
File.open()

File.write()

```
fileObject.write(param0, param1,...paramN)
```

Writes data to a file. The `write()` method converts each parameter to a string and then writes it to the file without separators. The file contents are buffered internally. The `File.flush()` method writes the buffer to the file on disk. When this method fails, it invokes the `application.onStatus()` event handler to report errors.

Note: *The user or process owner that the server runs under in the operating system must have write permissions or this call can fail.*

Availability
Flash Media Server 2

Parameters
`param0, param1,...paramN` Parameters to write to the file.

Returns
A boolean value indicating whether the write operation was successful (`true`) or not (`false`).

Example
The following example writes `"Hello world"` at the end of the `myFileObject` text file:

```
if (myFileObject.open( "text", "append") ) {
    myFileObject.write("Hello world");
}
```

File.writeAll()

```
fileObject.writeAll(array)
```

Takes an Array object as a parameter and calls the `File.writeln()` method on each element in the array. The file contents are buffered internally. The `File.flush()` method writes the buffer to the file on disk.

Note: *The user or process owner that the server runs under in the operating system must have write permissions or this call can fail.*

Availability
Flash Media Server 2

Parameters
`array` An Array object containing all the elements to write to the file.

Returns
A boolean value indicating whether the write operation was successful (`true`) or not (`false`).

File.writeByte()

```
fileObject.writeByte(number)
```

Writes a byte to a file. The file contents are buffered internally. The `File.flush()` method writes the buffer to the file on disk.

Note: *The user or process owner that the server runs under in the operating system must have write permissions or this call can fail.*

Availability
Flash Media Server 2

Parameters
`number` A number to write.

Returns
A boolean value indicating whether the write operation was successful (`true`) or not (`false`).

Example
The following example writes byte 65 to the end of the `myFileObject` file:

```
if (myFileObject.open("text","append")) {
    myFileObject.writeByte(65);
}
```

File.writeBytes()

```
fileObject.writeBytes(source, offset, length)
```

Writes `length` number of bytes to the `fileObject` from the `source` starting at the `offset` within `source`. This method throws an EOFError if `length` exceeds `source.length - offset`.

Availability
Flash Media Server 4

Parameters
`source` A ByteArray from which bytes to the `fileObject` are written.

`offset` An integer specifying an offset location within the ByteArray specifyed in the `source` parameter. This parameter is optional. The default value is 0.

`length` An integer specifying the number of bytes to write to the `fileObject`. This parameter is optional. The default value is `source.length - offset`.

Returns
Nothing.

File.writeln()

```
fileObject.writeln(param0, param1,...paramN)
```

Writes data to a file and adds a platform-dependent end-of-line character after outputting the last parameter. The file contents are buffered internally. The `File.flush()` method writes the buffer to the file on disk.

Note: *The user or process owner that the server runs under in the operating system must have write permissions or this call can fail.*

Availability
Flash Media Server 2

Parameters
`param0, param1,...paramN` Strings to write to the file.

Returns
A boolean value indicating whether the write operation was successful (`true`) or not (`false`).

Example
The following example opens a text file for writing and writes a line:

```
if (fileObj.open( "text", "append") ) {
    fileObj.writeln("This is a line!");
}
```

GroupSpecifier class

The GroupSpecifier class is used to construct the opaque strings called "groupspecs" to pass to NetStream and NetGroup constructors. A groupspec specifies an RTMFP peer-to-peer group, including the capabilities, restrictions, and authorizations of the member using the groupspec.By default, all capabilities are disabled, and peer-to-peer connections are allowed.

Neighbors within an RTMFP group may be introduced to each other in the following ways:

- Server channel automatic bootstrapping.

  When clients connect over an RTMFP connection, the server bootstraps them with peers who are members of the same group. To enable automatic bootstrapping, set `GroupSpecifier.serverChannelEnabled` to `true`.

- Explicit peerID exchange.

  Two or more peers pass their peerIDs to each other. There are any number of ways for peers to exchange peerIDs, from web services to telling each other verbally.

- LAN peer discovery

  Use the GroupSpecifier class for LAN peer discovery. LAN peer discovery allows a server-side RTMFP NetConnection and its NetStream and NetGroup objects to automatically locate peers and join a group on the current subnet. Peers cannot discover each other unless they're in the same group on the same subnet of the LAN. If peers with matching groupspecs are on different subnets, no error or other event is dispatched if they fail to discover each other.

  The following code shows how to enable LAN peer discovery in Server-Side ActionScript. The code would be written within an event callback function or in an RPC function.

```
var nc = new NetConnection();
// Protocol must be RTMFP
nc.connect("rtmfp://localhost/appname");
var gs = new GroupSpecifier("discovery-test");
// Must be enabled for LAN peer discovery to work
gs.ipMulticastMemberUpdatesEnabled = true;
// Multicast address over which to exchange peer discovery.
gs.addIPMulticastAddress("224.0.0.255:30000");
// Necessary to multicast over a NetStream.
gs.multicastEnabled = true;
var ns = new NetStream(nc, gs.toString());
```

GroupSpecifier constructor

`new GroupSpecifier(name:String)`

Creates a GroupSpecifier object. By default, all capabilities are disabled, and peer-to-peer connections are allowed.

Availability
Flash Media Server 4

Parameters
`name` A String specifying a name for the group. All members must use this name to join the group.

Returns
A GroupSpecifier object if successful. If the `name` parameter is missing or null, throws a JavaScript error.

Example
The following example creates a GroupSpecifier object called `groupSpecifier`. The group created with this object will have multicast and posting enabled. The group will also have the server channel enabled.

```
var groupSpecifier = new GroupSpecifier("com.example.someapp");
groupSpecifier.multicastEnabled = true;
groupSpecifier.postingEnabled = true;
groupSpecifier.serverChannelEnabled = true;
```

GroupSpecifier.addBootstrapPeer()

```
groupSpecifier.addBootstrapPeer(peerID)
```

Causes the associated NetStream or NetGroup to make an initial neighbor connection to the specified `peerID`.

Availability
Flash Media Server 4

Parameters
`peerID` A String. The peerID to which an initial neighbor connection should be made to bootstrap into the peer-to-peer mesh.

Returns
Nothing.

GroupSpecifier.addIPMulticastAddress()

```
groupSpecifier.addIPMulticastAddress(address, port)
```

Causes the associated NetStream or NetGroup to join the specified IP multicast group and listen to the specified UDP port.

If the `address` or `port` has an invalid format, a runtime error is raised.

Availability
Flash Media Server 4

Parameters
`address` A String specifying the address of the IPv4 or IPv6 multicast group to join, optionally followed by a colon (":") and the UDP port number. If specifying an IPv6 address and a port, the IPv6 address must be enclosed in square brackets, for example, `"224.0.0.254"`, `"224.0.0.254:30000"`, `"ff03::ffff"`, `"[ff03::ffff]:30000"`.

`port` A Number. The UDP port on which to receive IP multicast datagrams. If `port` is null, the UDP port must be specified in address. If not null, the UDP port must not be specified in address.

Returns
Nothing.

GroupSpecifier.authorizations()

```
groupSpecifier.authorizations()
```

Returns a string that represents passwords for IP multicast publishing and for posting. Append the string to an unauthorized groupspec to enable features for which passwords have been set.

Availability
Flash Media Server 4

Parameters
None.

Returns
A String.

GroupSpecifier.encodeBootstrapPeerIDSpec()

```
groupSpecifier.encodeBootstrapPeerIDSpec(peerID)
```

Encodes and returns a string that represents a bootstrap peerID. If you append the string to a groupspec, the associated NetStream or NetGroup makes an initial neighbor connection to the specified peerID.

Availability
Flash Media Server 4

Parameters
`peerID` A String. The peerID to which an initial neighbor connection should be made to bootstrap into the peer-to-peer mesh.

Returns
A String.

GroupSpecifier.encodeGroupspecDigest()

```
GroupSpecifier.encodeGroupspecDigest(groupspec:String)
```

A static method that encodes and returns the digest for the canonical portion of the supplied groupspec. Call this method to generate a groupspec digest to correlate Group join and leave events to a known Group.

Availability
Flash Media Server 4.5

Parameters
`groupspec` A String. The groupspec to encode a digest for.

Returns
The digest for the canonical portion of the supplied groupspec.

GroupSpecifier.encodeIPMulticastAddressSpec()

```
groupSpecifier.encodeIPMulticastAddressSpec(address, port)
```

Encodes and returns a string that represents an IP multicast socket address. If you append the string to a groupspec, the associated NetStream or NetGroup joins the specified IP multicast group and listens to the specified UDP port.

Throws a JavaScript error if the `address` or `port` parameters are invalid or missing.

Availability
Flash Media Server 4

Parameters
`address` A String. A String specifying the address of the IPv4 or IPv6 multicast group to join, optionally followed by a colon (":") and the UDP port number. If specifying an IPv6 address and a port, the IPv6 address must be enclosed in square brackets, for example, `"224.0.0.254"`, `"224.0.0.254:30000"`, `"ff03::ffff"`, `"[ff03::ffff]:30000"`.

`port` A Number. The UDP port on which to receive IP multicast datagrams. If `port` is null, the UDP port must be specified in address. If not null, the UDP port must not be specified in address.

Returns
A String.

GroupSpecifier.encodePostingAuthorization()

```
groupSpecifier.encodePostingAuthorization(password)
```

Encodes and returns a string that represents a posting password. When posting is password-protected, you can concatenate the string to a groupspec to enable posting.

Availability
Flash Media Server 4

Parameters
`password` A String. The password to encode, which must match the posting password (if set) to enable `NetGroup.post()`.

Returns
A String.

GroupSpecifier.encodePublishAuthorization()

```
groupSpecifier.encodePublishAuthorization(password)
```

Encodes and returns a string that represents a multicast publishing password. When multicast publishing is password-protected, you can concatenate the string to a groupspec to enable posting.

Availability
Flash Media Server 4

Parameters
`password` A String. The password to encode, which must match the publish password (if set) to enable `NetStream.publish()`.

Returns
A String.

GroupSpecifier.groupspecWithAuthorizations()

```
groupSpecifier.groupspecWithAuthorizations()
```

Returns the opaque groupspec string, including authorizations, that can be passed to the NetStream and NetGroup constructors.

Availability
Flash Media Server 4

Parameters
None.

Returns
A String.

GroupSpecifier.groupspecWithoutAuthorizations()

```
groupSpecifier.groupspecWithoutAuthorizations()
```

Returns the opaque groupspec string, without authorizations, that can be passed to the NetStream and NetGroup constructors.

Availability
Flash Media Server 4

Parameters
None.

Returns
A String.

GroupSpecifier.ipMulticastMemberUpdatesEnabled

```
groupSpecifier.ipMulticastMemberUpdatesEnabled
```

A Boolean value indicating whether or not information about group membership is exchanged on IP multicast sockets. IP multicast servers can send group membership updates to help bootstrap P2P meshes or heal partitions. Peers can send membership updates on the LAN to help bootstrap LAN P2P meshes and to inform on-LAN neighbors in global meshes that other on-LAN neighbors exist, which can improve P2P performance.

Availability
Flash Media Server 4

GroupSpecifier.makeUnique()

```
groupSpecifier.makeUnique()
```

Adds a strong pseudorandom tag to the groupspec to make it unique. The opaque groupspec string must then be passed to other potential members of the group. To join a group, a client must use this groupspec string.

Availability
Flash Media Server 4

Parameters
None.

Returns
Nothing.

GroupSpecifier.multicastEnabled

`groupSpecifier.multicastEnabled`

A Boolean value specifying whether or not streaming over a NetStream is enabled for the specified group. The default value is `false`.

Availability
Flash Media Server 4

GroupSpecifier.objectReplicationEnabled

`groupSpecifier.objectReplicationEnabled`

A Boolean value specifying whether or not Object Replication is enabled for the specified NetGroup. The default value is `false`.

Availability
Flash Media Server 4

GroupSpecifier.peerToPeerDisabled

`groupSpecifier.peerToPeerDisabled`

A Boolean value specifying whether or not peer-to-peer connections are disabled in this NetGroup or NetStream. The default value is `false`.

If peer-to-peer connections are disabled, it is guaranteed that no upstream bandwidth will be used by any member of the group because no neighbor connections will be made. If peer-to-peer connections are disabled, the peer-to-peer warning dialog is suppressed. This mode is only useful for sending and receiving multicast streams using pure IP multicast.

Availability
Flash Media Server 4

GroupSpecifier.postingEnabled

`groupSpecifier.postingEnabled`

A Boolean value specifying whether or not posting is enabled for the specified NetGroup. The default value is `false`. For information about posting, see `NetGroup.post()`.

Availability
Flash Media Server 4

GroupSpecifier.routingEnabled

`groupSpecifier.routingEnabled`

A Boolean value specifying whether or not the directed routing methods are enabled in the specified NetGroup. The default value is `false`.

Availability
Flash Media Server 4

GroupSpecifier.serverChannelEnabled

`groupSpecifier.serverChannelEnabled`

A Boolean value specifying whether or not members attempt to open a channel to the server for this group. The default value is `false`.

A channel to the server must be open before the server can provide supporting functions, such as bootstrapping, to group members. Supporting functions may or may not be provided over this channel depending on server configuration.

To use the server channel for automatic bootstrapping, set the `JoinLeaveEventsmode` attribute to `"All"` in the Application.xml file.

```
<GroupControl>
    <JoinLeaveEvents mode="All"/>
</GroupControl>
```

Availability
Flash Media Server 4

GroupSpecifier.setPostingPassword()

`groupSpecifier.setPostingPassword(password, salt)`

Specifies a password that is required to call `NetGroup.post()`.

Availability
Flash Media Server 4

Parameters
`password` A String. The password. If null, no password is required to post. The default value is null.

`salt` A String. Modifies the hash of the password to increase the difficulty of guessing it. For best security, this parameter should be set to a random value. The default value is null.

Returns
Nothing.

GroupSpecifier.setPublishPassword()

`groupSpecifier.setPublishPassword(password, salt)`

Specifies a password that is required to call `NetStream.publish()` to multicast publish into a group.

Availability
Flash Media Server 4

Parameters
`password` A String. The password. If null, no password is required. The default value is null.

`salt` A String. Modifies the hash of the password to increase the difficulty of guessing it. For best security, this parameter should be set to a random value. The default value is null.

Returns
Nothing.

GroupSpecifier.toString()

`groupSpecifier.toString()`

Identical to the `groupspecWithAuthorizations()` method. Convenience method to return the opaque groupspec string, including authorizations, to pass to NetStream and NetGroup constructors.

Availability
Flash Media Server 4

Parameters
None.

Returns
A String.

GroupControl

Flash Media Server 4.5

The GroupControl class represents an association between a remote peer and a NetGroup that it has joined.

Use the GroupControl class to distribute peer lookup requests across multiple servers.

For more information
Distribute peer introductions across servers

GroupControl.addNeighbor()

```
gc.addNeighbor(peerID:String);
```

Directs the remote peer to connect directly to the specified `peerID`, which must already be in this group, and adds it as a neighbor.

This method advises the remote peer to connect to the specified neighbor. The connection process is separate and happens only after this method is called.

Throws an error if any arguments are omitted or if arguments are the wrong type.

Available
Flash Media Server 4.5

Parameters
peerID A String. The peerID to connect to.

Returns
A Boolean value, `true` if the add neighbor request was dispatched to the remote peer, `false` otherwise.

GroupControl.addMemberHint()

```
gc.addMemberHint(peerID:String);
```

Directs the remote peer to add a record specifying that `peerID` is a member of the group. An immediate connection to it is attempted only if it is needed for the topology.

Throws an error if any arguments are omitted or if arguments are of the wrong type.

Available
Flash Media Server 4.5

Parameters
peerID A String. The peerID to add to the set of potential neighbors.

Returns
A Boolean value, `true` if the add member hint request was dispatched to the remote peer, `false` otherwise.

GroupControl.groupspecDigest

Read-only; The digest of the canonical groupspec, which securely identifies the group this client has joined.

Available
Flash Media Server 4.5

LoadVars class

The LoadVars class lets you send all the variables in an object to a specified URL and lets you load all the variables at a specified URL into an object. It also lets you send specific variables, rather than all variables, which can make your application more efficient. You can use the `LoadVars.onLoad()` handler to ensure that your application runs when data is loaded, and not before.

The LoadVars class works much like the XML class; it uses the `load()`, `send()`, and `sendAndLoad()` methods to communicate with a server. The main difference between the LoadVars class and the XML class is that LoadVars transfers ActionScript name-value pairs, rather than an XML Document Object Model (DOM) tree stored in the XML object. The LoadVars class follows the same security restrictions as the XML class.

Availability

Flash Media Server 2

Property summary

Property	Description
`LoadVars.contentType`	The MIME type sent to the server when you call the `LoadVars.send()` or `LoadVars.sendAndLoad()` method.
`LoadVars.loaded`	A boolean value that indicates whether a `LoadVars.load()` or `LoadVars.sendAndLoad()` operation has completed (`true`) or not (`false`).

Method summary

Method	Description
`LoadVars.addRequestHeader()`	Adds or changes HTTP request headers (such as Content-Type or SOAPAction) sent with `POST` actions.
`LoadVars.decode()`	Converts the query string to properties of the specified LoadVars object.
`LoadVars.getBytesLoaded()`	Returns the number of bytes loaded from the last or current `LoadVars.send()` or `LoadVars.sendAndLoad()` method call.
`LoadVars.getBytesTotal()`	Returns the number of total bytes loaded during all `LoadVars.send()` or `LoadVars.sendAndLoad()` method calls.
`LoadVars.load()`	Downloads variables from the specified URL, parses the variable data, and places the resulting variables in the LoadVars object that calls the method.
`LoadVars.send()`	Sends the variables in the specified object to the specified URL.
`LoadVars.sendAndLoad()`	Posts the variables in the specified object to the specified URL.
`LoadVars.toString()`	Returns a string containing all enumerable variables in the specified object, in the MIME content encoding *application/x-www-urlform-encoded*.

Event handler summary

Event handler	Description
`LoadVars.onData()`	Invoked when data has completely downloaded from the server or when an error occurs while data is downloading from a server.
`LoadVars.onHTTPStatus()`	Invoked when Adobe Media Server receives an HTTP status code from the server.
`LoadVars.onLoad()`	Invoked when a `LoadVars.send()` or `LoadVars.sendAndLoad()` operation has completed.

LoadVars constructor

```
new LoadVars()
```

Creates a LoadVars object. You can use the methods of the LoadVars object to send and load data.

Availability
Flash Media Server 2

Example
The following example creates a LoadVars object called `my_lv`:

```
var my_lv = new LoadVars();
```

LoadVars.addRequestHeader()

```
myLoadVars.addRequestHeader(header, headerValue)
```

Adds or changes HTTP request headers (such as Content-Type or SOAPAction) sent with `POST` actions. There are two possible use cases for this method: you can pass two strings, `header` and `headerValue`, or you can pass an array of strings, alternating header names and header values.

If multiple calls are made to set the same header name, each successive value replaces the value set in the previous call.

Availability
Flash Media Server 2

Parameters
`header` A string or an array of strings that represents an HTTP request header name.

`headerValue` A string that represents the value associated with `header`.

Example
The following example adds a custom HTTP header named SOAPAction with a value of `Foo` to the `my_lv` object:

```
var my_lv = new LoadVars();
my_lv.addRequestHeader("SOAPAction", "'Foo'");
```

The following example creates an array named headers that contains two alternating HTTP headers and their associated values. The array is passed as a parameter to the `addRequestHeader()` method.

```
var my_lv = new LoadVars();
var headers = ["Content-Type", "text/plain", "X-ClientAppVersion", "2.0"];
my_lv.addRequestHeader(headers);
```

The following example creates a new LoadVars object that adds a request header called FLASH-UUID. The header contains a variable that the server can check.

```
var my_lv = new LoadVars();
my_lv.addRequestHeader("FLASH-UUID", "41472");
my_lv.name = "Mort";
my_lv.age = 26;
my_lv.send("http://flash-mx.com/mm/cgivars.cfm", "_blank", "POST");
```

LoadVars.contentType

`myLoadVars.contentType`

The MIME type sent to the server when you call the `LoadVars.send()` or `LoadVars.sendAndLoad()` method. The default is *application/x-www-urlform-encoded*.

Availability
Flash Media Server 2

Example
The following example creates a LoadVars object and displays the default content type of the data that is sent to the server:

```
application.onConnect = function(client){
    this.acceptConnection(client);
    var my_lv = new LoadVars();
    trace(my_lv.contentType);
};

// Output to Live Log: application/x-www-form-urlencoded
```

LoadVars.decode()

`myLoadVars.decode(queryString)`

Converts the query string to properties of the specified LoadVars object. This method is used internally by the `LoadVars.onData()` event handler. Most users do not need to call this method directly. If you override the `LoadVars.onData()` event handler, you can explicitly call `LoadVars.decode()` to parse a string of variables.

Availability
Flash Media Server 2

Parameters
`queryString` A URL-encoded query string containing name-value pairs.

Example
The following example traces the three variables:

```
application.onConnect = function(client){
    this.acceptConnection(client);
    // Create a new LoadVars object.
    var my_lv = new LoadVars();
    //Convert the variable string to properties.
    my_lv.decode("name=Mort&score=250000");
    trace(my_lv.toString());
    // Iterate over properties in my_lv.
    for (var prop in my_lv) {
trace(prop+" -> "+my_lv[prop]);
    }
};
```

The following is output to the Live Log panel in the Administration Console:

```
name=Mort&score=250000
name -> Mort
score -> 250000
contentType -> application/x-www-form-urlencoded
loaded -> false
```

LoadVars.getBytesLoaded()

`myLoadVars.getByesLoaded()`

Returns the number of bytes loaded from the last or current `LoadVars.load()` or `LoadVars.sendAndLoad()` method call. The value of the `contentType` property does not affect the value of `getBytesLoaded()`.

Availability
Flash Media Server 2

Returns
A number.

See also
LoadVars.getBytesTotal()

LoadVars.getBytesTotal()

`myLoadVars.getBytesTotal()`

Returns the total number of bytes loaded into an object during all`LoadVars.load()` or `LoadVars.sendAndLoad()` `LoadVars.load()` or `LoadVars.sendAndLoad()` method calls. Each time a call to `load()` or `sendAndLoad()` is issued, the `getBytesLoaded()` method is reset, but the `getBytesTotal()` method continues to grow.

The value of the `contentType` property does not affect the value of `getBytesLoaded()`.

Availability
Flash Media Server 2

Returns
A number. Returns `undefined` if no load operation is in progress or if a load operation has not been initiated. Returns `undefined` if the number of total bytes can't be determined—for example, if the download was initiated but the server did not transmit an HTTP content length.

See also
`LoadVars.getBytesLoaded()`

LoadVars.load()

`myLoadVars.load(url)`

Downloads variables from the specified URL, parses the variable data, and places the resulting variables into the LoadVars object that calls the method. You can load variables from a remote URL or from a URL in the local file system; the same encoding standards apply to both.

Any properties in the `myLoadVars` object that have the same names as downloaded variables are overwritten. The downloaded data must be in the MIME content type and be *application/x-www-urlform-encoded*.

The `LoadVars.load()` method call is asynchronous.

Availability
Flash Media Server 2

Parameters
`url` A string indicating the URL from which to download variables.

Returns
A boolean value indicating success (`true`) or failure (`false`).

Example
The following code defines an `onLoad()` handler function that signals when data is returned:

```
application.onConnect = function(client){
    this.acceptConnection(client);
    var my_lv = new LoadVars();
    my_lv.onLoad = function(success) {
        if (success) {
            trace(this.toString());
        } else {
            trace("Error loading/parsing LoadVars.");
        }
    };
    my_lv.load("http://www.helpexamples.com/flash/params.txt");
};
```

LoadVars.loaded

`myLoadVars.loaded`

A boolean value that indicates whether a `LoadVars.load()` or `LoadVars.sendAndLoad()` operation has completed (`true`) or not (`false`).

Availability
Flash Media Server 2

Example
The following example loads a text file and writes information to the log file when the operation is complete:

```
var my_lv = new LoadVars();
my_lv.onLoad = function(success) {
trace("LoadVars loaded successfully: "+this.loaded);
};
my_lv.load("http://www.helpexamples.com/flash/params.txt");
```

See also
`LoadVars.onLoad()`

LoadVars.onData()

`myLoadVars.onData(src){}`

Invoked when data has completely downloaded from the server or when an error occurs while data is downloading from a server.

Availability

Flash Media Server 2

Parameters

`src` A string or `undefined`; the raw (unparsed) data from a `LoadVars.load()` or `LoadVars.sendAndLoad()` method call.

Details

This handler is invoked before the data is parsed and can be used to call a custom parsing routine instead of the one built in to Flash Player. The value of the `src` parameter that is passed to the function assigned to `LoadVars.onData()` can be either `undefined` or a string that contains the URL-encoded name-value pairs downloaded from the server. If the `src` parameter is `undefined`, an error occurred while downloading the data from the server.

The default implementation of `LoadVars.onData()` invokes `LoadVars.onLoad()`. You can override this default implementation by assigning a custom function to `LoadVars.onData()`, but `LoadVars.onLoad()` is not called unless you call it in your implementation of `LoadVars.onData()`.

Example

The following example loads a text file and traces the content when the operation is complete:

```
var my_lv = new LoadVars();
my_lv.onData = function(src) {
if (src == undefined) {
trace("Error loading content.");
return;
}
    trace(src);
};
my_lv.load("content.txt", my_lv, "GET");
```

LoadVars.onHTTPStatus()

`myLoadVars.onHTTPStatus(httpStatus){}`

Invoked when Adobe Media Server receives an HTTP status code from the server. This handler lets you capture and act on HTTP status codes.

Availability

Flash Media Server 2

Parameters

`httpStatus` A number; the HTTP status code returned by the server. For example, a value of 404 indicates that the server has not found a match for the requested URL. HTTP status codes can be found in sections 10.4 and 10.5 of the HTTP specification. (For more information, see the W3 website at www.w3.org.)

Details

The `onHTTPStatus()` handler is invoked before `onData()`, which triggers calls to `onLoad()` with a value of `undefined` if the load fails. After `onHTTPStatus()` is triggered, `onData()` is always triggered, whether or not you override `onHTTPStatus()`. To best use the `onHTTPStatus()` handler, you should write a function to catch the result of the `onHTTPStatus()` call; you can then use the result in your `onData()` and `onLoad()` handlers. If `onHTTPStatus()` is not invoked, this indicates that Adobe Media Server did not try to make the URL request.

If Adobe Media Server cannot get a status code, or if it cannot communicate with the server, the default value of 0 is passed to your ActionScript code.

Example

The following example shows how to use `onHTTPStatus()` to help with debugging. The example collects HTTP status codes and assigns their value and type to an instance of the LoadVars object. (Notice that this example creates the instance members `this.httpStatus` and `this.httpStatusType` at runtime.) The `onData()` handler uses these instance members to trace information about the HTTP response that can be useful in debugging.

```
var myLoadVars = new LoadVars();

myLoadVars.onHTTPStatus = function(httpStatus) {
    this.httpStatus = httpStatus;
    if(httpStatus < 100) {
        this.httpStatusType = "flashError";
    }
    else if(httpStatus < 200) {
        this.httpStatusType = "informational";
    }
    else if(httpStatus < 300) {
        this.httpStatusType = "successful";
    }
    else if(httpStatus < 400) {
        this.httpStatusType = "redirection";
    }
    else if(httpStatus < 500) {
        this.httpStatusType = "clientError";
    }
    else if(httpStatus < 600) {
        this.httpStatusType = "serverError";
    }
}

myLoadVars.onData = function(src) {
    trace(">> " + this.httpStatusType + ": " + this.httpStatus);
    if(src != undefined) {
        this.decode(src);
        this.loaded = true;
        this.onLoad(true);
    }
    else {
        this.onLoad(false);
    }
}

myLoadVars.onLoad = function(success) {}

myLoadVars.load("http://weblogs.macromedia.com/mxna/flashservices/getMostRecentPosts.cfm");
```

LoadVars.onLoad()

```
myLoadVars.onLoad(success){}
```

Invoked when a `LoadVars.load()` or `LoadVars.sendAndLoad()` operation has completed. If the variables load successfully, the `success` parameter is `true`. If the variables were not received, or if an error occurred in receiving the response from the server, the `success` parameter is `false`.

If the `success` parameter is `true`, the `myLoadVars` object is populated with variables downloaded by the `LoadVars.load()` or `LoadVars.sendAndLoad()` operation, and these variables are available when the `onLoad()` handler is invoked.

Availability
Flash Media Server 2

Parameters
`success` A boolean value indicating whether the `LoadVars.load()` operation ended in success (`true`) or failure (`false`).

Example
The following example creates a new LoadVars object, attempts to load variables into it from a remote URL, and prints the result:

```
myLoadVars = new LoadVars();
myLoadVars.onLoad = function(result){
    trace("myLoadVars load success is " + result);
}
myLoadVars.load("http://www.someurl.com/somedata.txt");
```

LoadVars.send()

```
myLoadVars.send(url [, target, method])
```

Sends the variables in the `myLoadVars` object to the specified URL. All enumerable variables in the `myLoadVars` object are concatenated into a string that is posted to the URL by using the HTTP `POST` method.

The MIME content type sent in the HTTP request headers is the value of `LoadVars.contentType`.

Availability
Flash Media Server 2

Parameters
`url` A string; the URL to which to upload variables.

`target` A File object. If you use this optional parameter, any returned data is output to the specified File object. If this parameter is omitted, the response is discarded.

`method` A string indicating the `GET` or `POST` method of the HTTP protocol. The default value is `POST`. This parameter is optional.

Returns
A boolean value indicating success (`true`) or failure (`false`).

See also

`LoadVars.sendAndLoad()`

LoadVars.sendAndLoad()

```
myLoadVars.sendAndLoad(url, target[, method ])
```

Posts the variables in the `myLoadVars` object to the specified URL. The server response is downloaded and parsed as variable data, and the resulting variables are placed in the `target` object. Variables are posted in the same way as `LoadVars.send()`. Variables are downloaded into `target` in the same way as `LoadVars.load()`.

Availability

Flash Media Server 2

Parameters

`url` A string; the URL to which to upload variables.

`target` The LoadVars object that receives the downloaded variables.

`method` A string; the `GET` or `POST` method of the HTTP protocol. The default value is `POST`. This parameter is optional.

Returns

A boolean value indicating success (`true`) or failure (`false`).

LoadVars.toString()

```
myLoadVars.toString()
```

Returns a string containing all enumerable variables in `myLoadVars`, in the MIME content encoding *application/x-www-form-urlencoded*.

Availability

Flash Media Server 2

Returns

A string.

Example

The following example instantiates a new `LoadVars()` object, creates two properties, and uses `toString()` to return a string containing both properties in URL-encoded format:

```
var my_lv = new LoadVars();
my_lv.name = "Gary";
my_lv.age = 26;
trace (my_lv.toString());
//output: age=26&name=Gary
```

Log class

The Log class lets you create a Log object that can be passed as an optional parameter to the constructor for the WebService class.

Availability
Flash Media Server 2

Event handler summary

Event handler	Description
`Log.onLog()`	Invoked when a log message is sent to a log.

Log constructor

```
new Log([logLevel][, logName])
```

Creates a Log object that can be passed as an optional parameter to the constructor for the WebService class.

Availability
Flash Media Server 2

Parameters
`logLevel` One of the following values (if not set explicitly, the level defaults to `Log.BRIEF`):

Value	Description
`Log.BRIEF`	Primary life cycle event and error notifications are received.
`Log.VERBOSE`	All life cycle event and error notifications are received.
`Log.DEBUG`	Metrics and fine-grained events and errors are received.

`logName` An optional parameter that can be used to distinguish between multiple logs that are running simultaneously to the same output.

Returns
A Log object.

Example
The following example creates a new instance of the Log class:

```
newLog = new Log();
```

Log.onLog()

```
myLog.onLog(message){}
```

Invoked when a log message is sent to a log.

Availability
Flash Media Server 2

Parameters
`message` A log message.

MulticastStreamInfo class

Flash Media Server 4.5

The MulticastStreamInfo class specifies various Quality of Service (QoS) statistics related to a NetStream object's underlying RTMFP peer-to-peer and IP Multicast stream transport. A MulticastStreamInfo object is returned by the `NetStream.multicastInfo` property.

Properties that return numbers represent totals computed from the beginning of the multicast stream. These types of properties include the number of media bytes sent or the number of media fragment messages received. Properties that are rates represent a snapshot of the current rate averaged over a few seconds. These types of properties include the rate at which a local node is receiving data.

To see a list of values contained in the MulticastStreamInfo object, use the `MulticastStreamInfo.toString()` method.

MulticastStreamInfo.bytesPushedToIPMulticast

`msi.bytesPushedToIPMulticast`

Read-only; Specifies the the total count of media bytes sent by the local node to IP Multicast.

MulticastStreamInfo.bytesReceivedFromIPMulticast

`msi.bytesReceivedFromIPMulticast`

Read-only; Specifies the total count of media bytes received by the local node from IP Multicast.

Availability
Flash Media Server 4

MulticastStreamInfo.bytesReceivedFromServer

`msi.bytesReceivedFromServer`

Read-only; Specifies the total count of media bytes received by the local node from the server.

Availability
Flash Media Server 4

MulticastStreamInfo.bytesRequestedByPeers

`msi.bytesRequestedByPeers`

Read-only; Specifies the total count of media bytes sent by the local node to peers in response to requests from those peers for specific fragments.

Availability
Flash Media Server 4

MulticastStreamInfo.bytesRequestedFromPeers

`msi.bytesRequestedFromPeers`

Read-only; Specifies the total count of media bytes received by the local node from peers that were specifically requested by the local node.

Availability
Flash Media Server 4

MulticastStreamInfo.fragmentsPushedFromPeers

`msi.fragmentsPushedFromPeers`

Read-only; Specifies the total count of media fragment messages received by the local node that were pushed by peers.

Availability
Flash Media Server 4

MulticastStreamInfo.fragmentsPushedToIPMulticast

`msi.fragmentsPushedToIPMulticast`

Read-only; Specifies the total count of media fragments sent by the local node to IP Multicast.

Availability
Flash Media Server 4

MulticastStreamInfo.fragmentsPushedToPeers

`msi.fragmentsPushedToPeers`

Read-only; Specifies the total count of media fragment messages pushed by the local node to peers.

Availability
Flash Media Server 4

MulticastStreamInfo.fragmentsReceivedFromIPMulticast

`msi.fragmentsReceivedFromIPMulticast`

Read-only; Specifies the total count of media fragment messages received by the local node from IP Multicast.

Availability
Flash Media Server 4

MulticastStreamInfo.fragmentsReceivedFromServer

`msi.fragmentsReceivedFromServer`

Read-only; Specifies the total count of media fragment messages received by the local node from the server.

Availability
Flash Media Server 4

MulticastStreamInfo.fragmentsRequestedByPeers

`msi.fragmentsRequestedByPeers`

Read-only; Specifies the total count of media fragment messages sent by the local node to peers in response to requests from those peers for specific fragments.

Availability
Flash Media Server 4

MulticastStreamInfo.fragmentsRequestedFromPeers

`msi.fragmentsRequestedFromPeers`

Read-only; Specifies the total count of media fragment messages received by the local node from peers that were specifically requested by the local node.

Availability
Flash Media Server 4

MulticastStreamInfo.receiveControlBytesPerSecond

`msi.receiveControlBytesPerSecond`

Read-only; Specifies the data rate of control overhead messages received by the local node from peers in bytes per second.

Availability
Flash Media Server 4

MulticastStreamInfo.receiveDataBytesPerSecond

`msi.receiveDataBytesPerSecond`

Read-only; Specifies the total rate at which media data is being received by the local node from peers, the server, and over IP multicast, in bytes per second.

Availability
Flash Media Server 4

MulticastStreamInfo.receiveDataBytesPerSecondFromIPMulticast

`msi.receiveDataBytesPerSecondFromIPMulticast`

Read-only; Specifies the rate at which media data is being received by the local node from IP Multicast in bytes per second.

Availability
Flash Media Server 4

MulticastStreamInfo.receiveDataBytesPerSecondFromServer

`msi.receiveDataBytesPerSecondFromServer`

Read-only; Specifies the rate at which media data is being received by the local node from the server in bytes per second.

Availability
Flash Media Server 4

MulticastStreamInfo.sendControlBytesPerSecond

```
msi.sendControlBytesPerSecond
```

Read-only; Specifies the total data rate of control overhead messages sent by the local node to peers and the server in bytes per second.

Availability
Flash Media Server 4

MulticastStreamInfo.sendControlBytesPerSecondToServer

```
msi.sendControlBytesPerSecondToServer
```

Read-only; Specifies the data rate of control overhead messages sent by the local node to the server in bytes per second.

Availability
Flash Media Server 4

MulticastStreamInfo.sendDataBytesPerSecond

```
msi.sendDataBytesPerSecond
```

Read-only; Specifies the rate at which media data is being sent by the local node to peers in bytes per second.

Availability
Flash Media Server 4

MulticastStreamInfo.sendDataBytesPerSecondToIPMulticast

```
msi.sendDataBytesPerSecondToIPMulticast
```

Read-only; Specifies the rate at which media data is being sent by the local node to IP Multicast in bytes per second.

Availability
Flash Media Server 4

MulticastStreamInfo.toString()

```
ns.toString()
```

Returns a text value listing the properties of the MulticastStreamInfo object.

Availability
Flash Media Server 4

MulticastStreamIngest class

Flash Media Server 4.5

Use the MulticastStreamIngest class to ingest RTMFP multicast streams and convert the multicast data units to messages that a Stream object can use. To create a MulticastStreamIngest instance, call `NetGroup.getMulticastStreamIngest()`. Then call `Stream.playFromGroup()` to play the stream from the group.

You can also use Server-Side ActionScript to:

- Record the Stream object.
- Deliver the Stream object to clients over RTMP/T/S/E
- Package the stream for delivery using HTTP Dynamic Streaming and HTTP Live Streaming.

For more information
Multicasting

MulticastStreamIngest.ingesting

`mcsi.ingesting`

Read-only; A Boolean property that is `true` if the target multicast stream is bound and currently being ingested; otherwise `false`.

Availability
Flash Media Server 4.5

MulticastStreamIngest.multicastInfo

`mcsi.multicastInfo`

Read-only; A MulticastStreamInfo object whose properties contain statistics about the stream quality of service.

Availability
Flash Media Server 4.5

MulticastStreamIngest.multicastWindowDuration

`mcsi.multicastWindowDuration`

The duration, in seconds, of the peer-to-peer multicast reassembly window. The default value is 8.

Availability
Flash Media Server 4.5

MulticastStreamIngest.multicastPushNeighborLimit

`mcsi.multicastPushNeighborLimit`

The maximum number of peers to which to push multicast media. The default value is 4.

Even though the script is pulling data from a group in order to transform the data, the peer is still in the group and expected to pass along the data to other members of the group. If you don't want to pass along the data, set this value to 0.

Availability
Flash Media Server 4.5

MulticastStreamIngest.close()

```
mcsi.close()
```

Stops ingesting the source multicast stream.

Availability
Flash Media Server 4.5

Parameters
None.

Returns
None.

MulticastStreamIngest.onStatus()

```
mcsi.onStatus = function(infoObject){}
```

Invoked when a MulticastStreamIngest status change occurs.

Availability
Flash Media Server 4.5

Parameters
infoObject An Object with the following properties:

- `infoObject.code`
- `infoObject.code.level`
- `infoObject.fragmentsLost`—a Number indicating the number of multicast data units lost.

The `code` and `level` properties provides information about the status change. All values are of type String:

code	level	Description
`"NetStream.MulticastStream.GapNotify"`	`"status"`	One or more multicast data units have been lost from the stream being ingested. The `fragmentsLost` property contains the amount that were lost.
`"NetStream.MulticastStream.Reset"`	`"status"`	This is the same as the `"NetStream.MulticastStream.Reset"` event that is dispatched by NetStream. The multicast stream has been reset to a different stream published with the same name in the same Flash group.
`"NetGroup.MulticastStream.PublishNotify"`	`"status"`	This is the same as the `"NetGroup.MulticastStream.PublishNotify"` event that is dispatched by NetGroup. However, this event is dispatched only for the multicast stream that is being targeted for ingest rather than for any multicast stream within the Flash group
`"NetGroup.MulticastStream.UnpublishNotify"`	`"status"`	This is the same as the `"NetGroup.MulticastStream.UnpublishNotify"` event that is dispatched by NetGroup. However, this event is dispatched only for the multicast stream that is being ingested rather than for any multicast stream within the Flash group.

NetConnection class

The server-side NetConnection class lets you create a two-way connection between an Adobe Media Server application instance and an application server, another Adobe Media Server, or another Adobe Media Server application instance on the same server. You can use NetConnection objects to create powerful applications; for example, you can get weather information from an application server or share an application load with other servers that are running Adobe Media Server or application instances.

Availability
Flash Communication Server 1

Property summary

Property	Description
NetConnection.farID	Read-only. A String indetifying the RTMFP identify of the Adobe Media Server instance to which this Adobe MediaAdobe Media Server Instance is connected.
NetConnection.farNonce	Read-only. A String unique to this connection. This value appears to another server as its NetConnection.nearNonce value.
NetConnection.isConnected	Read-only; a boolean value indicating whether a connection has been made.
NetConnection.nearID	Read-only. A String indentifier of this Flash Player or Adobe AIR instance for this NetConnection instance.
NetConnection.nearNonce	Read-only. A String unique to this connection. This value appears to another server as its NetConnection.farNonce value.
NetConnection.objectEncoding	The Action Message Format (AMF) version used to pass binary data between two servers.
NetConnection.rtmfpBindAddresses	An Array of Strings representing the specific address or addresses that the NetConnection binds locally when it opens its RTMFP protocol stack.
NetConnection.rtmfpEndpointName	The endpoint name for the local RTMFP protocol stack.
NetConnection.uri	Read-only; a string indicating the `URI` parameter of the `NetConnection.connect()` method.

Method summary

Method	Description
NetConnection.addHeader()	Adds a context header to the Action Message Format (AMF) packet structure.
NetConnection.call()	Invokes a command or method on another Adobe Media Server or an application server to which the application instance is connected.
NetConnection.close()	Closes the connection with the server.
NetConnection.connect()	Connects to another Adobe Media Server or to a Flash Remoting server such as Adobe ColdFusion.

Event handler summary

Event handler	Description
NetConnection.onStatus()	Invoked every time the status of the NetConnection object changes.

NetConnection constructor

```
new NetConnection()
```

Creates a new instance of the NetConnection class.

Availability

Flash Communication Server 1.

Returns

A NetConnection object.

Example

The following example creates a new instance of the NetConnection class:

```
newNC = new NetConnection();
```

NetConnection.addHeader()

```
nc.addHeader(name, mustUnderstand, object)
```

Adds a context header to the Action Message Format (AMF) packet structure. This header is sent with every future AMF packet. If you call `addHeader()` by using the same name, the new header replaces the existing header, and the new header persists for the duration of the NetConnection object. To remove a header, call `addHeader()` and pass it the name of the header to remove and an undefined object.

Availability

Flash Communication Server 1

Parameters

`name` A string; identifies the header and the ActionScript object data associated with it.

`mustUnderstand` A boolean; `true` indicates that the server must understand and process this header before it handles any of the following headers or messages.

`object` An Object.

Example

The following example creates a new NetConnection instance, `nc`, and connects to an application at web server www.foo.com that is listening at port 1929. This application dispatches the service /blag/SomeCoolService. The last line of code adds a header called `foo`.

```
nc=new NetConnection();
nc.connect("http://www.foo.com:1929/blag/SomeCoolService");
nc.addHeader("foo", true, new Foo());
```

NetConnection.call()

```
nc.call(methodName, [resultObj [, p1, ..., pN])
```

Invokes a command or method on another Adobe Media Server or an application server to which the application instance is connected. The `NetConnection.call()` method on the server works the same way as the `NetConnection.call()` method on the client: it invokes a command on a remote server.

Note: *To call a method on a client from a server, use the `Client.call()` method.*

Availability

Flash Communication Server 1

Parameters

`methodName` A string indicating a method specified in the form `"[objectPath/]method"`. For example, the `someObj/doSomething` command tells the remote server to invoke the `clientObj.someObj.doSomething()` method, with all the `p1, ..., pN` parameters. If the object path is missing, `clientObj.doSomething()` is invoked on the remote server.

`resultObj` An Object. This optional parameter is used to handle return values from the server. The result object can be any object that you defined and can have two defined methods to handle the returned result: `onResult()` and `onStatus()`. If an error is returned as the result, `onStatus()` is invoked; otherwise, `onResult()` is invoked.

p1, ..., pN Optional parameters that can be of any ActionScript type, including a reference to another ActionScript object. These parameters are passed to the `methodName` parameter when the method is executed on the remote application server.

Returns

For RTMP connections, returns a boolean value of `true` if a call to `methodName` is sent to the client; otherwise, `false`. For application server connections, it always returns `true`.

Example

The following example uses RTMP to execute a call from one Adobe Media Server to another Adobe Media Server. The code makes a connection to the App1 application on server 2 and then invokes the `Sum()` method on server 2:

```
nc1.connect("rtmp://server2.mydomain.com/App1", "svr2",);
nc1.call("Sum", new Result(), 3, 6);
```

The following Server-Side ActionScript code is on server 2. When the client is connecting, this code checks to see whether it has a parameter that is equal to `svr1`. If the client has that parameter, the `Sum()` method is defined so that when the method is called from svr1, svr2 can respond with the appropriate method:

```
application.onConnect = function(clientObj){
    if(arg1 == "svr1"){
        clientObj.Sum = function(p1, p2){
            return p1 + p2;
        }
    }
    return true;
};
```

The following example uses an Action Message Format (AMF) request to make a call to an application server. This allows Adobe Media Server to connect to an application server and then invoke the `quote()` method. The Java™ adaptor dispatches the call by using the identifier to the left of the dot as the class name and the identifier to the right of the dot as a method of the class.

```
nc = new NetConnection;
nc.connect("http://www.xyz.com/java");
nc.call("myPackage.quote", new Result());
```

NetConnection.close()

```
nc.close()
```

Closes the connection with the server. After you close the connection, you can reuse the NetConnection instance and reconnect to an old application or connect to a new one.

Note: *The `NetConnection.close()` method has no effect on HTTP connections.*

Availability

Flash Communication Server 1

NetConnection.connect()

```
nc.connect(URI, [p1, ..., pN])
```

Call `NetConnection.connect()` to do any of the following:

- (Flash Media Server 1.0) Connect over HTTP to an application server, such as Adobe® ColdFusion®, running a Flash Remoting gateway.

  Note: *You cannot use HTTP to connect to another Adobe Media Server or to media assets.*

- (Flash Media Server 3.0) Multi-point publishing over RTMP/S. Connect over RTMP/S to another Adobe Media Server and publish a stream to the server.
- (Flash Media Server 4.0) Multi-point publishing over RTMFP. Connect over RTMFP to an application on the same server or on another Adobe Media Server and publish a stream to the server.

  Create a server-side NetConnection, connect to the target server with an RTMFP URI (for example, `"rtmfp://ams.example.com/myapp"`). Then create a NetStream without passing a GroupSpecifier string as a constructor argument. This technique is traditional multi-point publishing but over an RTMFP connection.

- (Flash Media Server 4.0) Publish a live stream into a NetGroup. Connect over RTMFP to the current application or to an application on another Adobe Media Server.

  After you create an RTMFP NetConnection, use a GroupSpecifier to create a NetGroup and a NetStream. The server application joins a group and becomes a peer in the group mesh. A single client can join multiple groups. Once connected, you can interact with other peers in the group and publish into the group. Use this technique to ingest a live stream and publish it into a group.

  Note: *Server-Side ActionScript doesn't support opening an RTMFP NetConnection directly to a client peer (`DIRECT_CONNECTIONS` in the client API).*

- (Flash Media Server 4.5) Pass the string `"rtmfp:"` to create a serverless network endpoint for RTMFP communication. This mode can be used only for RTMFP groups; it cannot be used for direct connections.

  This mode is more fault-tolerate and supports the ability to cluster servers via RTMFP with no single point of failure. See Distribute introductions across servers.

It is good practice to write an `application.onStatus()` callback function and check the `NetConnection.isConnected` property for RTMP connections to see whether a successful connection was made. For Action Message Format (AMF) connections, check `NetConnection.onStatus()`.

Availability

Flash Communication Server 1

Parameters

`URI` A string indicating a URI to connect to. URI has the following format:

```
[protocol://]host[:port]/appName[/instanceName]
```

The following are legal URIs:

```
http://appserver.mydomain.com/webapp
rtmp://ams.mydomain.com/realtimeapp
rtmps://ams.mydomain.com/secureapp
rtmp://localhost/realtimeApp
rtmp:/realtimeApp
rtmfp://ams.mydomain.com/p2papp
rtmfp://
```

`p1, ..., pN` Optional parameters that can be of any ActionScript type, including references to other ActionScript objects. These parameters are sent as connection parameters to the `application.onConnect()` event handler for RTMP connections. For AMF connections to application servers, RTMP parameters are ignored.

Returns

For RTMP and RTMFP connections, a boolean value of `true` for success; otherwise, `false`. For AMF connections to application servers, `true` is always returned.

Example

The following example creates an RTMP connection to an application instance on Adobe Media Server:

```
nc = new NetConnection();
nc.connect("rtmp://ams.example.com/vod/instance1");
```

NetConnection.farID

`nc.farId`

Read-only. A String indentifying the RTMFP identify of the Adobe Media Server instance to which this Adobe Media Server is connected. This property is meaningful only for RTMFP connections. The value of this property is available only after an RTMFP connection is established.

Availability

Flash Media Server 4

See also

`NetConnection.nearID`

NetConnection.farNonce

`nc.farNonce`

Read-only. A String unique to this connection. This value appears to another server as its `NetConnection.nearNonce` value. This value is defined for RTMFP, RTMPE, and RTMPTE connections.

Availability

Flash Media Server 4

See also

"`NetConnection.nearNonce`" on page 102

NetConnection.isConnected

`nc.isConnected`

Read-only; a boolean value indicating whether a connection has been made. It is set to `true` if there is a connection to the server. It's a good idea to check this property value in an `onStatus()` callback function. This property is always `true` for AMF connections to application servers.

Availability

Flash Communication Server 1

Example

The following example uses `NetConnection.isConnected` in an `onStatus()` handler to check whether a connection has been made:

```
nc = new NetConnection();
nc.connect("rtmp://tc.foo.com/myApp");
nc.onStatus = function(infoObj){
    if (info.code == "NetConnection.Connect.Success" && nc.isConnected){
        trace("We are connected");
    }
};
```

NetConnection.nearID

`nc.nearId`

A String.

Read-only. A String indentifier of this Adobe Media Server instance for this NetConnection session. This property is meaningful only for RTMFP connections.

Availability

Flash Media Server 4

See also

`NetConnection.farID`

NetConnection.nearNonce

`nc.nearNonce`

Read-only. A String unique to this connection. This value appears to another server as its `NetConnection.farNonce` value. This value is defined for RTMFP, RTMPE, and RTMPTE connections.

Availability

Flash Media Server 4

See also

"`NetConnection.farNonce`" on page 101

NetConnection.objectEncoding

`nc.objectEncoding`

The Action Message Format (AMF) version used to pass binary data between two servers. The possible values are 3 (ActionScript 3.0 format) and 0 (ActionScript 1.0 and ActionScript 2.0 format). The default value is 3. When Adobe Media Server acts as a client trying to connect to another server, the encoding of the client should match the encoding of the remote server.

Flash Media Server versions 1 and 2 support AMF0. Flash Media Server versions 3 and later support AMF0 and AMF3.

The value of `objectEncoding` is determined dynamically according to the following rules when the server receives a `NetConnection.onStatus()` event with the code property `NetConnection.Connect.Success`:

- If the `onStatus()info` object contains an `objectEncoding` property, its value is used.
- If the `onStatus()info` object does not contain an `objectEncoding` property, 0 is assumed even if the connecting server has set `objectEncoding` to 3.

- Once the NetConnection instance is connected, the `objectEncoding` property is read-only.

These rules turn Flash Media Server 3 into an AMF0 client when it connects to a remote Flash Media Server version 2 or earlier (which only support AMF0).

Note: *The server always serializes data in AMF0 while executing Flash Remoting functions.*

Availability
Flash Media Server 3

NetConnection.onStatus()

```
nc.onStatus = function(infoObject) {}
```

Invoked every time the status of the NetConnection object changes. For example, if the connection with the server is lost in an RTMP connection, the `NetConnection.isConnected` property is set to `false`, and `NetConnection.onStatus()` is invoked with a status message of `NetConnection.Connect.Closed`. For AMF connections, `NetConnection.onStatus()` is used only to indicate a failed connection. Use this event handler to check for connectivity.

Availability
Flash Communication Server 1

Parameters
`infoObject` An Object with properties that provide information about the status of a NetConnection information object. This parameter is optional, but it is usually used. The NetConnection information object contains the following properties:

Property	Meaning
`code`	A string identifying the event that occurred.
`description`	A string containing detailed information about the code. Not every information object includes this property.
`level`	A string indicating the severity of the event.

The following table contains the `code` and `level` property values and their meanings:

Code	Level	Meaning
`NetConnection.Call.Failed`	`error`	The `NetConnection.call()` method was not able to invoke the server-side method or command.
`NetConnection.Connect.AppShutdown`	`error`	The application has been shut down (for example, if the application is out of memory resources and must shut down to prevent the server from crashing) or the server has shut down.
`NetConnection.Connect.Closed`	`status`	The connection was closed successfully.
`NetConnection.Connect.Failed`	`error`	The connection attempt failed.

Code	Level	Meaning
`NetConnection.Connect.Rejected`	`error`	The client does not have permission to connect to the application, or the application name specified during the connection attempt was not found on the server. This information object also has an `application` property that contains the value returned by `application.rejectConnection()`.
`NetConnection.Connect.Success`	`status`	The connection attempt succeeded.
`NetConnection.Proxy.NotResponding`	`error`	The proxy server is not responding. See the ProxyStream class.

Example

The following example defines a function for the `onStatus()` handler that outputs messages to indicate whether the connection was successful:

```
nc = new NetConnection();
nc.onStatus = function(info){
    if (info.code == "NetConnection.Connect.Success") {
        _root.gotoAndStop(2);
    }   else {
        if (! nc.isConnected){
            _root.gotoAndStop(1);
        }
    }
};
```

NetConnection.rtmfpBindAddresses

`nc.rtmfpBindAddresses`

An Array of Strings representing the specific address or addresses that the NetConnection binds locally when it opens its RTMFP protocol stack. This value is ignored for all other protocols. If you assign an `NetConnection.rtmfpEndpointName` to a NetConnection, assign this property also so that other peers can interact with the instance at a known IP address and port.

When assigning an Array of address Strings to this property, each String must specify an IPv4 or IPv6 address to bind and listen on, optionally followed by a colon (":") and a UDP port number. To specify an IPv6 address and a port, enclose the IPv6 address in square brackets. If no port value is specified, the next available system port is bound. Upon successful bind, the String value in the property representing the address is updated to include the ephemeral port number that has been bound.

By default, the RTMFP protocol stack binds `"0.0.0.0"` if connecting to an IPv4 RTMFP server, or `"::"` if connecting to an IPv6 server when the computer, network and Adobe Media Server support IPv6. These values direct the RTMFP protocol stack to listen on either all available IPv4 or IPv6 interfaces, but not both.To bind and listen on IPv4 and IPv6 interfaces concurrently, assign an Array of addresses such as: `["0.0.0.0:31000", "[::]:31000"]`. In this example, all local IPv4 and IPv6 interfaces are bound on port 31000. Use non-wildcard address values to selectively bind specific interfaces rather than all of them.

This property can be read at any time but can only be set when the NetConnection is in a disconnected state and no connect attempt is in progress.

This property throws an error if any arguments within the assigned Array are of the wrong type or missing, address values are improperly formatted, or the NetConnection is connected or is in the process of connecting.

Binding happens after the `NetConnection.connect()` method is called. If a bind attempt fails, a `NetConnection.Connect.Failed` event is dispatched, and details of the bind failure are logged to the application log.

Availability
Flash Media Server 4.5

Example
The following example tells the NetConnection to bind and listen on any IPv4 interfaces at port 31000:

```
nc.rtmfpBindAddresses = ["0.0.0.0:31000];
```

NetConnection.rtmfpEndpointName

`nc.rtmfpEndpointName`

The endpoint name for the local RTMFP protocol stack; ignored for all other protocols. The default value is null, which is ignored by the local RTMFP protocol stack. This property can be read at any time but cannot be assigned while the NetConnection is connected or in the process of connecting.

This property throws an error if the assigned name is of the wrong type or missing, or if the NetConnection is currently connected or in the process of connecting.

Availability
Flash Media Server 4.5

Example
The following example assigns an RTMFP endpoint name:

```
nc.rtmfpEndpointName = "bootstrap-peer";
```

NetConnection.uri

`nc.uri`

Read-only; a string indicating the `URI` parameter of the `NetConnection.connect()` method. This property is set to `null` before a call to `NetConnection.connect()` or after a call to `NetConnection.close()`.

Availability
Flash Communication Server 1

NetGroup class

Instances of the NetGroup class represent membership in an RTMFP group. Use this class to do the following:

- **Monitor Quality of Service**. The `info` property contains a NetGroupInfo object whose properties provide QoS statistics for this group.
- **Posting**. Call `NetGroup.post()` to broadcast ActionScript messages to all members of a group.
- **Direct routing**. Call `sendToNearest()`, `sendToNeighbor()`, and `sendToAllNeighbors()` to send a short data message to a specific member of a peer-to-peer group. The source and the destination do not need to have a direct connection.
- **Ingest a multicast stream**. Call `getMulticastStreamIngest()` to ingest a multicast RTMFP stream from a group.

For more information, see Building peer-assisted networking applicationsin the *Adobe Media Server Developer's Guide*.


For information about peer-assisted networking, see Basics of P2P in Flash by Adobe Evangelist Tom Krcha. For information about using groups with peer-assisted networking, see Social media experiences with Adobe Media Server and RTMFP, also by Tom Krcha.


For information about the technical details behind peer-assisted networking, see P2P on the Flash Platform with RTMFP by Adobe Computer Scientist Matthew Kaufman.

NetGroup class constructor

```
netGroup = new NetGroup(connection, groupspec)
```

Constructs a NetGroup on the specified NetConnection object and joins it to the group specified by the `groupspec` parameter. Use the GroupSpecifier class to build the `groupspec` string. In Server-Side ActionScript, you can pass a `groupspec` String or a GroupSpecifier object as the `groupspec` argument. The NetConnection must use the RTMFP protocol.

Throws an Error if either argument is missing or null, if the NetGroup fails to join the group specified in the `groupspec` parameter, or if the protocol is not RTMFP.

The status messages are dispatched to "`NetGroup.onStatus()`" on page 110.

Availability

Flash Media Server 4

Parameters

`connection` A NetConnection object. The protocol must be RTMFP.

`groupspec` To publish or play a stream in a peer-to-peer multicast group, specify a groupspec string or a GroupSpecifier object. To create a groupspec string, call `GroupSpecifier.toString()`.

Example

The following example passes a GroupSpecifier object as the `groupspec` parameter:

```
netGroup = new NetGroup(myConnection, myGroupSpecifier);
```

The following example passes a GroupSpecifier string as the `groupspec` parameter:

```
netGroup = new NetGroup(myConnection, myGroupSpecifier.toString());
```

NetGroup.addMemberHint()

```
netGroup.addMemberHint(peerID)
```

Manually adds a record specifying that peerID is a member of the Group. An immediate connection to it is attempted only if it is needed for the topology.

Availability

Flash Media Server 4

Parameters

`peerID` A String. The peerID to add to the set of potential neighbors.

Returns
A Boolean value. If successful, returns `true`. Otherwise, returns `false`.

NetGroup.addNeighbor()

`netGroup.addMemberHint(peerID)`

Manually adds a neighbor by immediately connecting directly to the specified peerID, which must already be in this Group. This direct connection may later be dropped if it is not needed for the topology.

Availability
Flash Media Server 4

Parameters
`peerID` A String. The peerID to which to connect.

Returns
A Boolean value. If successful, returns `true`. Otherwise, returns `false`.

NetGroup.addPermanentNeighborByName()

`netGroup.addPermanentNeighborByName(rtmfpEndpointName:String, addresses:Array);`

Manually adds a neighbor, by RTMFP endpoint name, which must already be in this group via a NetGroup instance constructed using a NetConnection that has been assigned the target RTMFP endpoint name. Unlike `addNeighbor()`, this direct connection is permanent.

Throws an error if any argument is of the wrong type or missing, or if address values within the addresses argument are incorrectly formatted.

Availability
Flash Media Server 4.5

Parameters
`rtmfpEndpointName` A String. The name assigned to the RTMFP protocol stack at the target peer. See `NetConnection.rtmfpEndpointName`.

`addresses` An Array of address Strings specifying the IPv4 or IPv6 addresses where the target peer is running, followed by a colon (":") and port number. If specifying an IPv6 address and a port, enclose the IPv6 address in square brackets.

Returns
A Boolean value. If successful, returns `true`. Otherwise, returns `false`.

Example
The following example adds a neighbor with the RTMFP endpoint name `bootstrap-peer`, using a known address where the peer is running:

```
ng.addPermanentNeighborByName("bootstrap-peer", [bootstrapPeerAddress]);
```

NetGroup.close()

`netGroup.close()`

Disconnect from the group and close this NetGroup. This NetGroup is not usable after calling this method.

Availability
Flash Media Server 4

Parameters
None.

Returns
None.

NetGroup.convertPeerIDToGroupAddress()

`netGroup.convertPeerIDToGroupAddress(peerID)`

Converts a peerID to a group address to pass to the `sendToNearest()` method.

Availability
Flash Media Server 4

Parameters
`peerID` A String. The peerID to convert

Returns
A String. The group address for the peerID..

NetGroup.getMulticastStreamIngest()

`netGroup.getMulticastStreamIngest(sourceStreamName)`

A factory function used to construct a MulticastStreamIngest class. Call this function to ingest a multicast stream from an RTMFP group.

It's a good idea to call `getMulticastStreamIngest()` in response to a `"NetGroup.MulticastStream.PublishNotify event"`. Alternately, you could decide how long to wait for receipt of multicast stream bytes and use `setInterval()` to close down the ingest attempt.

This class throws an error if the `stream` parameter is not a String. If the stream name is incorrect (but is a String), it will not throw an error. For example, if you specify a `sourceStreamName` that isn't currently being published into the Group, you will not receive an error. Multicast streams within a Group do not necessarily have a sole source or owner, so there's no single peer to contact to determine whether the stream name is valid or not.

If this is an issue for your application, consider checking whether the returned MulticastStreamIngest object is ingesting on an interval and logging a failure after an application-defined timeout.

Availability
Flash Media Server 4.5

Parameters

`sourceStreamName` A String. The name of the source multicast stream.

Returns

If a MulticastStreamIngest instance already exists for this group with this stream name, it returns the existing one, otherwise it creates a new MulticastStreamIngest instance.

Example

The following example is pseudocode that provides a high-level description of how to ingest a multicast stream, convert it into a Stream object, record it, and play it.

```
// First, set up Stream instance that will play (and record) the multicast ingest.
var stream = Stream.get("mp4:multicast-ingest.f4v");

// Next, set up a server-side NetConnection and NetGroup to join the Flash Group
// where the desired multicast stream is being published.
var nc = new NetConnection();
nc.onStatus = function(info) {
    if (info.code == "NetConnection.Connect.Success") {
        ng = new NetGroup(nc, groupspec);
        ng.onStatus = ngStatusHandler;
    }
};
nc.connect("rtmfp://<ams-introduction-server>...");

// Handle NetGroup status events; this simple example handles only the initial join.
function ngStatusHandler(info) {
    if (info.code == "NetGroup.Connect.Success") {
        // As soon as we've successfully joined the Group,
        // attempt to start ingesting a multicast stream.
        // Assume we know the stream name we want to ingest (sourceStreamName).
        ingest = ng.getMulticastStreamIngest(sourceStreamName);
    }
}

// At some later point (or directly in the ngStatusHandler above),
// play the multicast ingest.
stream.playFromGroup(ingest);

// The stream can be recorded locally.
stream.record();
...
stream.record(false); // And recording stopped.

// To stop playback of a multicast stream, pass the Boolean false.
stream.playFromGroup(false);
```

See also

Ingest, convert, and record a multicast stream

NetGroup.estimatedMemberCount

```
groupSpecifier.estimatedMemberCount
```

Read-only; A Number specifying the estimated number of members of the group, based on local neighbor density and assuming an even distribution of group addresses.

Availability
Flash Media Server 4

NetGroup.info

```
groupSpecifier.info
```

Read-only; a NetGroupInfo object whose properties provide Quality of Service statistics related to this NetGroup's RTMFP peer-to-peer data transport.

Availability
Flash Media Server 4

NetGroup.localCoverageTo

```
groupSpecifier.localCoverageTo
```

Read-only; Specifies the end of the range of group addresses for which this node is the "nearest" and responsible. The range is specified in the increasing direction along the group address ring mod 2256.

Availability
Flash Media Server 4

NetGroup.localCoverageFrom

```
groupSpecifier.localCoverageFrom
```

Read-only; Specifies the start of the range of group addresses for which this node is the "nearest" and responsible. The range is specified in the increasing direction along the group address ring mod 2256.

Availability
Flash Media Server 4

NetGroup.onStatus()

```
netGroup.onStatus = function(infoObject){})
```

Invoked every time a status change or error occurs in a NetGroup object.

Availability
Flash Media Server 4

Parameters
`infoObject` An Object with `code` and `level` properties that provide information about the status of a NetGroup call. Both properties are strings.

Code property	Level property	Description
`"NetGroup.Connect.Failed"`	`"error"`	The creation of or connection to a NetGroup has failed. For example, this code is sent if there is an error in the GroupSpecifier.
`"NetGroup.Connect.Success"`	`"status"`	A NetGroup was created successfully.
`"NetGroup.LocalCoverage.Notify"`	`"status"`	Sent when a portion of the group address space for which this node is responsible changes.
`"NetGroup.MulticastStream.PublishNotify"`	"status"	A NetStream has started publishing into a group.
`"NetGroup.MulticastStream.UnpublishNotify"`	"status"	A NetStream has stopped publishing into a group.
`"NetGroup.Neighbor.Connect"`	`"status"`	Sent when a neighbor connects to this node. There are two additional properties: `e.info.neighbor`, a String specifying the group address of the neighbor. `e.info.peerID`, a String specifying the peerID of the neighbor.
`"NetGroup.Neighbor.Disconnect"`	`"status"`	Sent when a neighbor disconnects to this node. There are two additional properties: `e.info.neighbor`, a String specifying the group address of the neighbor. `e.info.peerID`, a String specifying the peerID of the neighbor.
`"NetGroup.Posting.Notify"`	`"status"`	Dispatched when the group receives a message from the `NetGroup.post()` method. There are two additional properties: `info.message`, an Object containing the message, and `info.messageID`, a String containing the message's messageID.
`"NetGroup.SendTo.Notify"`	`"status"`	Dispatched when a message sent directly to this node is received. The `info.message` property is an Object containing the message. The `info.from` property is a String specifying the `groupAddress` from which the message was received. The `info.fromLocal` property is a Boolean value. The value is `true` if the message was sent by this node (the local node is the node closest to the destination group address), and `false` if the message was received from a different node. To implement recursive routing, if `info.fromLocal` is `false`, call `NetGroup.sendToNearest()` to resend the message.

Example

```
var netGroup = new NetGroup(nc, myGroupSpecifier);
netGroup.onStatus = function(info){
    if (info.code == "NetGroup.Connect.Success"){
        trace("Successful NetGroup connection");
    }
}
```

NetGroup.post()

```
netGroup.post(message)
```

If authorized, sends the message to all other members of the group. All messages must be unique; a message that is identical to one posted earlier might not be propagated.

This method returns the messageID for this message, or null on error. The messageID is the hex of the SHA256 of the raw bytes of the serialization of the message.

This method sends `"NetGroup.Posting.Notify"` to `NetStream.onStatus()` with two additional properties: `e.info.message`, an Object containing the message, and `e.info.messageID`, a String containing the message's messageID.

For more information, see Post messages to a groupin the *Adobe Media Server Developer's Guide*.

Availability
Flash Media Server 4

Parameters
`message` An Object. The message to send to all other members of the group. The message can be an Object, an int, a Number, or a String.

Returns
A String. The messageID of the message if posted, or null on error.

NetGroup.receiveMode

`groupSpecifier.receiveMode`

Specifies the routing receive mode for this node. Use a property of the NetGroupReceiveMode class. The value can be either `NetGroupReceiveMode.EXACT` or `NetGroupReceiveMode.NEAREST`. See the "NetGroupReceiveMode class" on page 115 for more information.

Availability
Flash Media Server 4

NetGroup.removePermanentNeighborByName()

`netGroup.removePermanentNeighborByName(rtmfpEndpointName:String);`

Manually removes the "permanent" status for a neighbor by RTMFP endpoint name. This method call does not cause an existing neighbor connection to be dropped immediately. However, if either end chooses to drop the connection at a future point the drop is allowed.

Throws an error if any argument is of the wrong type or missing.

Availability
Flash Media Server 4.5

Parameters
`rtmfpEndpointName` A String. The name assigned to the RTMFP protocol stack at the target peer. See `NetConnection.rtmfpEndpointName`.

Returns
A Boolean value. If successful, returns `true`. Otherwise, returns `false`.

Example
The following example removes the "permanent" status of a neighbor with the RTMFP endpoint name `bootstrap-peer`:

```
ng.removePermanentNeighborByName("bootstrap-peer");
```

NetGroup.sendToAllNeighbors()

```
netGroup.sendToAllNeighbors(message)
```

Sends a message to all neighbors. Returns `NetGroupSendResult.SENT` if at least one neighbor was selected.

When a node receives a message, a `"NetGroup.SendTo.Notify"` is sent to the `NetStream.onStatus()` method.

Use this method to route messages directly to a peer, also called "direct routing". See Route messages directly to a peer.

Availability
Flash Media Server 4

Parameters
`message` An Object. The message to send.

Returns
A String. A property of the "NetGroupSendResult class" on page 117 indicating the success or failure of the send.

NetGroup.sendToNearest()

```
netGroup.sendToNearest(message, groupAddress)
```

Sends a message to the neighbor (or local node) nearest to the specified group address. Considers neighbors from the entire ring. Returns `NetGroupSendResult.SENT` if the message was successfully sent toward its destination.

When a node receives a message, `"NetGroup.SendTo.Notify"` is sent to the `NetStream.onStatus()` method.

Use this method to route messages directly to a peer, also called "direct routing". See Route messages directly to a peer.

Availability
Flash Media Server 4

Parameters
`message` An Object. The message to send.

`groupAddress` A String. The group address toward which to route the message.

Returns
A String. A property of the "NetGroupSendResult class" on page 117 indicating the success or failure of the send.

NetGroup.sendToNeighbor()

```
netGroup.sendToNeighbor(message, sendMode)
```

Sends a message to the neighbor specified by the `sendMode` parameter. Returns `NetGroupSendResult.SENT` if the message was successfully sent to the requested destination.

When a node receives a message, a `"NetGroup.SendTo.Notify"` is sent to the `NetStream.onStatus()` method.

Use this method to route messages directly to a peer, also called "direct routing". See Route messages directly to a peer.

Availability
Flash Media Server 4

Parameters
`message` An Object. The message to send.

`sendMode` A String. A property of enumeration class NetGroupSendMode specifying the neighbor to which to send the message.

Returns
A String. A property of the "NetGroupSendResult class" on page 117 indicating the success or failure of the send.

NetGroupInfo class

The NetGroupInfo class specifies various Quality of Service (QoS) statistics related to a NetGroup object's RTMFP peer-to-peer data transport. The `NetGroup.info` property returns a NetGroupInfo object which is a snapshot of the current QoS state.

NetGroupInfo.objectReplicationReceiveBytesPerSecond

`NetGroupInfo.objectReplicationReceiveBytesPerSecond`

Read-only; Specifies the rate at which the local node receives objects from peers, in bytes per second.

Availability
Flash Media Server 4

NetGroupInfo.objectReplicationSendBytesPerSecond

`NetGroupInfo.objectReplicationSendBytesPerSecond`

Read-only; Specifies the rate at which objects are being copied from the local node to peers, in bytes per second.

Availability
Flash Media Server 4

NetGroupInfo.postingReceiveControlBytesPerSecond

`NetGroupInfo.postingReceiveControlBytesPerSecond`

Read-only; Specifies the rate at which the local node is receiving posting control overhead messages from peers, in bytes per second.

Availability
Flash Media Server 4

NetGroupInfo.postingReceiveDataBytesPerSecond

`NetGroupInfo.postingReceiveDataBytesPerSecond`

Read-only; Specifies the rate at which the local node is receiving posting data from peers, in bytes per second.

Availability
Flash Media Server 4

NetGroupInfo.postingSendControlBytesPerSecond

`NetGroupInfo.postingSendControlBytesPerSecond`

Read-only; Specifies the rate at which the local node is sending posting control overhead messages to peers, in bytes per second.

Availability
Flash Media Server 4

NetGroupInfo.postingSendDataBytesPerSecond

`NetGroupInfo.postingSendDataBytesPerSecond`

Read-only; Specifies the rate at which the local node is sending posting data to peers, in bytes per second.

Availability
Flash Media Server 4

NetGroupInfo.routingReceiveBytesPerSecond

`NetGroupInfo.routingReceiveBytesPerSecond`

Read-only; Specifies the rate at which the local node is receiving directed routing messages from peers, in bytes per second.

Availability
Flash Media Server 4

NetGroupInfo.routingSendBytesPerSecond

`NetGroupInfo.routingSendBytesPerSecond`

Read-only; Specifies the rate at which the local node is sending directed routing messages to peers, in bytes per second.

Availability
Flash Media Server 4

NetGroupReceiveMode class

The NetGroupReceiveMode class is an enumeration of constant values used for the `NetGroup.receiveMode` property.

NetGroupReceiveMode.EXACT

`NetGroupReceiveMode.EXACT`

Specifies that this node considers itself "nearest" for any `NetGroup.sendToNearest()` call only if the `groupAddress` parameter passed to `NetGroup.sendToNearest()` matches this node's group address exactly. This value is the default setting.

If you want to enable distributed routing behavior, set this value to `NetGroupReceiveMode.NEAREST`. With this value set, a node waits for its connectivity to stabilize before participating in the directed routing mesh.

Availability
Flash Media Server 4

NetGroupReceiveMode.NEAREST

`NetGroupReceiveMode.NEAREST`

Specifies that this node accepts local messages from neighbors that send messages to group addresses that don't match this node's address exactly. Messages are received when this node is nearest among all neighbors whose receive mode is `NetGroupReceiveMode.NEAREST`. Distance is measured between addresses on the ring mod 2256.

Availability
Flash Media Server 4

NetGroupReplicationStrategy class

The NetGroupReceiveMode class is an enumeration of constant values used for the `NetGroup.receiveMode` property.

NetGroupReplicationStrategy.LOWEST_FIRST

`NetGroupReplicationStrategy.LOWEST_FIRST`

Specifies that when fetching objects from a neighbor to satisfy a want, the objects with the lowest index numbers are requested first.

Availability
Flash Media Server 4

NetGroupReplicationStrategy.RAREST_FIRST

`NetGroupReplicationStrategy.RAREST_FIRST`

Specifies that when fetching objects from a neighbor to satisfy a want, the objects with the fewest replicas among all the neighbors are requested first.

Availability
Flash Media Server 4

NetGroupSendMode class

The NetGroupSendMode class is an enumeration of constant values used for the `NetGroup.sendToNeighbor()` method.

NetGroupSendMode.NEXT_DECREASING

`NetGroupSendMode.NEXT_DECREASING`

Specifies the neighbor with the nearest group address in the decreasing direction.

Availability
Flash Media Server 4

NetGroupSendMode.NEXT_INCREASING

`NetGroupSendMode.NEXT_INCREASING`

Specifies the neighbor with the nearest group address in the increasing direction.

Availability
Flash Media Server 4

NetGroupSendResult class

The NetGroupSendResult class is an enumeration of constant values used for the return value of the NetGroup directed routing methods.

NetGroupSendResult.ERROR

`NetGroupSendResult.ERROR`

Indicates an error occurred (such as no permission) when using a directed routing method.

Availability
Flash Media Server 4

NetGroupSendResult.NO_ROUTE

`NetGroupSendResult.NO_ROUTE`

Indicates no neighbor could be found to route the message toward its requested destination.

Availability
Flash Media Server 4

NetGroupSendResult.SENT

`NetGroupSendResult.SENT`

Indicates that a route was found for the message and it was forwarded toward its destination.

Availability
Flash Media Server 4

NetStream class

There are two main use cases for the NetStream class: publishing a stream to a remote Adobe Media Server Professional and publishing a stream to an RTMFP group.

A NetStream object is a channel inside a NetConnection object. Use the NetStream class to publish streams one-way through a NetConnection object. Unlike a client-side NetStream object, a server-side NetStream object can only publish streams; it cannot subscribe to a publishing stream or play a recorded stream.

Publish a stream to a remote server

Note: *This use case is available in Adobe Media Server Professional.*

Use the NetStream class to scale live broadcasting applications to support more clients. Adobe Media Server Professional can support only a certain number of subscribing clients. To increase that number, you can use the NetStream class to move traffic to remote servers while still maintaining only one client-to-server connection.

Note: *To copy a stream to a remote server over RTMFP, use the following steps and pass an RTMFP protocol to the* `NetConnection.connect()` *method.*

The following steps describe the workflow for publishing a stream to a remote Adobe Media Server Professional:

1. To create a NetConnection, call the NetConnection constructor:

   ```
   var nc = net NetConnection();
   ```

2. To connect to an application on a remote Adobe Media Server Professional, call the `connect()` method:

   ```
   nc.connect("rtmp://servername/appname/appinstancename");
   ```

   Note: *You cannot use RTMPT, RTMPE, or RTMPTE when connecting to a remote server.*

3. To create a stream over the NetConnection, call the NetStream constructor and pass the NetConnection:

   ```
   var ns = new NetStream(nc);
   ```

4. To publish a stream to a remote server, call `publish()` and pass a name for the stream.

   ```
   ns.publish("mystream");
   ```

5. To subscribe to this stream, clients connect to the same application on the remote server. To play the stream, call the `play()` method and pass the stream name. The following is client-side code:

   ```
   mynetconnection.connect("rtmp://servername/appname/appinstancename");
   // Check for a "NetConnection.Connect.Success" message, then call play().
   mynetstream.play("mystream");
   ```

Publish a stream to an RTMFP group

Note: *This use case is available in Flash Media Server 4.*

The following steps describe the workflow for publishing a stream to an RTMFP group:

1. To create a NetConnection, call the NetConnection constructor:

   ```
   var nc = net NetConnection();
   ```

2 To connect to the server acting as the introducer for the RTMFP group, call the `connect()` method and connect to the server-side application:

```
nc.connect("rtmfp://localhost/appname/appinstancename");
```

Note: *Remember to use the RTMFP protocol.*

3 To create a stream over the NetConnection, call the NetStream constructor and pass the NetConnection and a GroupSpecifier:

```
var ns = new NetStream(nc, groupspecifier);
```

4 To publish a stream to a group, call `publish()` and pass a name for the stream. Stream data is routed to the RTMFP group, it is not copied at the server.

```
ns.publish("mystream");
```

5 To subscribe to this stream, clients call the `play()` method and pass the stream name. Clients receive the stream from the group, not from the server. The following is client-side code:

```
mynetstream.play("mystream");
```

Availability

Flash Media Server 3

RTMFP groups are supported in Flash Media Server 4

Property summary

Property	Description
`NetStream.bufferTime`	Read-only; indicates the number of seconds assigned to the buffer by the `NetStream.setBufferTime()` method.
`NetStream.time`	Read-only; indicates the number of seconds the stream has been publishing.

Method summary

Method	Description
`NetStream.attach()`	Attaches a data source to the NetStream object.
`NetStream.publish()`	Publishes a stream to a remote server.
`NetStream.send()`	Broadcasts a data message over a stream.
`NetStream.setBufferTime()`	Sets the size of the outgoing buffer in seconds.

Event handler summary

Event handler	Description
`NetStream.onStatus()`	Invoked every time a status change or error occurs in a NetStream Object.

NetStream class constructor

```
ns = new NetStream(connection[, groupspec])
```

Creates a stream that can be used for publishing (sending) data through the specified NetConnection object. You can create multiple streams that run simultaneously over the same connection.

To publish a stream through an RTMFP NetConnection to a group, pass a `groupspecifier` argument.

To publish a stream through an RTMFP NetConnection to a remote server, pass a connection argument. The NetConnection must use an RTMFP URI. Do not pass a groupspecifier argument.

Note: *You cannot use the server-side NetStream class to play a stream.*

Availability
Flash Media Server 3

RTMFP groups are available in Flash Media Server 4

Parameters
`connection` A NetConnection object.

`groupspec` (Optional) To publish or play a stream in a peer-to-peer multicast group, specify a groupspec string or a GroupSpecifier object. To create a groupspec string, call `GroupSpecifier.toString()`.

Returns
A NetStream object if successful; otherwise, `null`.

Example
```
nc = new NetConnection();
nc.connect("rtmp://xyz.com/myapp");
ns = new NetStream(nc);
```

NetStream.attach()

```
ns.attach(stream)
```

Attaches a data source to the NetStream object.

Availability
Flash Media Server 3

Parameters
`stream` A Stream object. If you pass `false`, the attached Stream object detaches from the NetStream object.

Returns
A boolean value. If the attached object is a valid data source, `true`; otherwise, `false`.

Example
```
myStream = Stream.get("foo");
ns = new NetStream(nc);
ns.attach(myStream);
```

NetStream.bufferTime

```
ns.bufferTime
```

Read-only; indicates the number of seconds assigned to the buffer by the `NetStream.setBufferTime()` method.

Availability
Flash Media Server 3

NetStream.farID

`ns.farID`

(Read-only) A String identifier of the far end connected to this NetStream instance. Always returns the `groupspec` value used to construct the NetStream object.

Availability
Flash Media Server 4

NetStream.multicastAvailabilitySendToAll

`ns.multicastAvailabilitySendToAll`

A Boolean value specifying whether peer-to-peer multicast fragment availability messages are sent to all peers simultaneously once every `multicastAvailabilityUpdatePeriod`. The default value is `false`.

Availability
Flash Media Server 4

NetStream.multicastAvailabilityUpdatePeriod

`ns.multicastAvailabilityUpdatePeriod`

A Number specifying the interval in seconds between messages sent to peers informing them that the local node has new peer-to-peer multicast media fragments available.

If a successful RTMFP NetConnection has not been made, the default value is 0. If a successful RTMFP NetConnection has been made, the default value is 0.1. Possible values are numbers in the range [0.0, 4294967.295]. Values outside this range are compressed to fit. Negative value assignments are treated as a 0 assignment. Values larger than the upper limit for the property are truncated to the upper limit for assignment. No error is raised or warning logged when this implicit argument coercion is performed.

Availability
Flash Media Server 4

NetStream.multicastFetchPeriod

`ns.multicastFetchPeriod`

A Number specifying the time in seconds between when the local node first learns of the availability of a peer-to-peer multicast media fragment and when it attempts to fetch it from a peer.

If a successful RTMFP NetConnection has not been made, the default value is 0. If a successful RTMFP NetConnection has been made, the default value is 2.5. Possible values are numbers in the range [0.0, 4294967.295]. Values outside this range are compressed to fit. Negative value assignments are treated as a 0 assignment. Values larger than the upper limit for the property are truncated to the upper limit for assignment. No error is raised or warning logged when this implicit argument coercion is performed.

Availability
Flash Media Server 4

NetStream.multicastInfo

`ns.multicastInfo`

Read-only; A MulticastStreamInfo object whose properties contain statistics about the quality of service.

Availability
Flash Media Server 4

NetStream.multicastPushNeighborLimit

`ns.multicastPushNeighborLimit`

A Number specifying the maximum number of peers to which to push multicast media.

If a successful RTMFP NetConnection has not been made, the default value is 0. If a successful RTMFP NetConnection has been made, the default value is 4. Possible values are numbers in the range [0, 4294967295]. Values outside this range are compressed to fit. Negative value assignments are treated as a 0 assignment. Values larger than the upper limit for the property are truncated to the upper limit for assignment. No error is raised or warning logged when this implicit argument coercion is performed.

Availability
Flash Media Server 4

NetStream.multicastRelayMarginDuration

`ns.multicastRelayMarginDuration`

A Number specifying the duration in seconds that peer-to-peer multicast data remains available to send to peers that request it beyond the `multicastWindowDuration`.

If a successful RTMFP NetConnection has not been made, the default value is 0. If a successful RTMFP NetConnection has been made, the default value is 2. Possible values are numbers in the range [0.0, 4294967.295]. Values outside this range are compressed to fit. Negative value assignments are treated as a 0 assignment. Values larger than the upper limit for the property are truncated to the upper limit for assignment. No error is raised or warning logged when this implicit argument coercion is performed.

Availability
Flash Media Server 4

NetStream.multicastWindowDuration

`ns.multicastWindowDuration`

A Number specifying the duration in seconds of the peer-to-peer multicast reassembly window.

If a successful RTMFP NetConnection has not been made, the default value is 0. If a successful RTMFP NetConnection has been made, the default value is 8. Possible values are numbers in the range [0.0, 4294967.295]. Values outside this range are compressed to fit. Negative value assignments are treated as a 0 assignment. Values larger than the upper limit for the property are truncated to the upper limit for assignment. No error is raised or warning logged when this implicit argument coercion is performed.

Availability
Flash Media Server 4

NetStream.nearNonce

`ns.nearNonce`

(Read-only) A String chosen substantially by this end of the stream, unique to this connection. For RTMFP Group streaming this value is the empty string.

Availability
Flash Media Server 4

NetStream.onStatus()

`ns.onStatus = function(infoObject){})`

Invoked every time a status change or error occurs in a NetStream object.

The remote server can accept or reject a call to `NetStream.publish()`.

Availability
Flash Media Server 3

Parameters
`infoObject` An Object with `code` and `level` properties that provide information about the status of a NetStream call. Both properties are strings.

Code property	Level property	Description
`"NetStream.Connect.Success"`	`"status"`	Dispatched when a NetStream is created successfully. In Server-Side ActionScript, this status is not sent to `NetConnection.onStatus()`.
`"NetStream.Connect.Failed"`	`"error"`	Dispatched when NetStream creation or connection fails (for example, if there is an error in the GroupSpecifier).
`"NetStream.MulticastStream.Reset"`	"status"	A multicast subscription has changed focus to a different stream published with the same name in the same group. Local overrides of multicast stream parameters are lost. Reapply the local overrides or the new stream's default parameters will be used.
`"NetStream.Play.Failed"`	`"error"`	In Server-Side ActionScript, a NetStream can publish a stream to a multicast group; it cannot play a stream.
`"NetStream.Publish.BadName"`	`"error"`	An attempt was made to publish to a stream that is already being published by someone else.
`"NetStream.Publish.Failed"`	`"error"`	A call to `NetStream.publish()` was attempted and failed. For example, permission was denied.
`"NetStream.Publish.Start"`	`"status"`	An attempt to publish was successful.
"`NetStream.Record.DiskQuotaExceeded`"	`"error"`	An attempt to record a stream failed because the disk quota was exceeded. For more information, see `Stream.record()`.
`"NetStream.Record.Failed"`	`"error"`	An attempt to record a stream failed.
`"NetStream.Record.NoAccess"`	`"status"`	An attempt was made to record a read-only stream.

Code property	Level property	Description
`"NetStream.Record.Start"`	`"status"`	Recording was started.
`"NetStream.Record.Stop"`	`"status"`	Recording was stopped.
`"NetStream.Unpublish.Success"`	`"status"`	An attempt to stop publishing a stream was successful.

Example

```
ns = new NetStream(nc);
ns.onStatus = function(info){
    if (info.code == "NetStream.Publish.Start"){
        trace("It is now publishing");
    }
ns.publish("foo", "live");
}
```

NetStream.publish()

```
ns.publish(name, howToPublish)
```

Publishes a stream to a remote server. Check the status in the `NetStream.onStatus()` handler to make sure that the remote server has accepted the publisher. If the stream has been published by another client, the `publish()` call can fail when it reaches the remote server. In this case, the remote server sends a status message of `"NetStream.Publish.BadName"` to the NetStream.onStatus() method.

Availability

Flash Media Server 3

Parameters

`name` A string identifying the stream to publish. If you pass `false`, the stream stops publishing. Use the following syntax:

File format	Syntax
FLV	`ns.publish("filename")`
MP3	`ns.publish("mp3:filename")` `ns.publish("id3:filename")`

MP4	`ns.publish("mp4:filename")`
	`ns.publish("mp4:filename.mp4")`
	`ns.publish("mp4:filename.f4v")`

`howToPublish` An optional string specifying how to publish the stream. Valid values are `"record"`, `"append"`, `"appendWithGap"`, and `"live"`. The default value is `"live"`. If you omit this parameter or pass `"live"`, live data is published but not recorded. If a file with this name already exists on the remote server, it is deleted.

Note: *If the file is read-only, live data is published and the file is not deleted.*

If you pass `"record"`, the stream is published and the data is recorded to a new file. If the file exists, it is overwritten. If you pass `"append"`, the stream is published and the data is appended to the existing stream specified by `name`. If no file is found, it is created.

If you pass `"appendWithGap"`, additional information about time coordination is passed to help the server determine the correct transition point when dynamic streaming. For more information about using "`appendWithGap`", see Using DVR with dynamic streaming.

The server stores recorded files in the streams subfolder of the application's folder, for example, *RootInstall*/applications/sampleapplication/_definst_/streams. The recorded file has the filename passed in the `name` parameter. For example, `NetStream.publish ("mp4:streamname.f4v", "record")` creates the file streamname.f4v.

Example

```
application.onPublish = function(client, myStream){
    nc = new NetConnection();
    nc.connect("rtmp://example.com/myApp");
    ns = new NetStream(nc);
    ns.attach(myStream);
    ns.publish(myStream.name, "live");
};
```

The following example shows how to record an F4V file on the remote server.

```
application.onPublish = function(client, myStream){
    nc = new NetConnection();
    nc.connect("rtmp://example.com/myApp");
    ns = new NetStream(nc);
    ns.attach(myStream);
    ns.publish("mp4:" + myStream.name, "record");
};
```

NetStream.send()

```
ns.send(handlerName, [p1, ..., pN])
```

Broadcasts a data message over a stream.

Availability

Flash Media Server 3

Parameters

`handlerName` A string that identifies the name of the handler to receive the message.

`p1, ..., pN` Optional parameters of any type. They are serialized and sent over the connection. The receiving handler receives them in the same order.

Returns

A boolean value; `true` if the data message is dispatched; otherwise, `false`.

Example

The following client-side code broadcasts the message `"Hello world"` to the `foo` handler function on each client that is connected to `myApp`:

```
nc = new NetConnection();
nc.connect("rtmp://xyz.com/myApp");
ns = new NetStream(nc);
ns.send("foo", "Hello world");
```

NetStream.setBufferTime()

```
ns.setBufferTime(bufferTime)
```

Sets the size of the outgoing buffer in seconds. If publishing, it controls the buffer in the local server.

Availability
Flash Media Server 3

Parameters
`bufferTime` A number indicating the size of the outgoing buffer in seconds.

Example

```
nc = new NetConnection();
nc.connect("rtmp://xyz.com/myApp");
ns = new NetStream(nc);
ns.setBufferTime(2);
```

NetStream.setIPMulticastPublishAddress()

```
ns.setIPMulticastPublishAddress(address, port)
```

Sets the native IP multicast address to which a stream publishes. To clear the address, pass null in the `address` parameter. If the `port` argument is null, you must specify the port within the `address` argument. If either the `address` or `port` values are invalid an Error is raised.

By default, a call to `publish()` does not use native IP multicast. Attempting to change the value while the NetStream is publishing raises an Error.

This method is for RTMFP connections only. If the NetStream is not publishing over a NetConnection that uses the RTMFP protocol, or if the NetStream wasn't constructed with a GroupSpecifier, any assigned value is ignored.

Availability
Flash Media Server 4

Parameters
`address` A String. The address of the IPv4 or IPv6 multicast group to publish to, optionally followed by a colon (":") and the UDP port number, or null to clear the existing value and disable native IP multicast. If specifying an IPv6 address and a port, the IPv6 address must be enclosed in square brackets. For example, `"224.0.0.254"`, `"224.0.0.254:30000"`, `"ff03::ffff"`, `"[ff03::ffff]:30000"`.

`port` A Number. The UDP port on which to send native IP multicast datagrams. If `port` is null, the UDP port must be specified in the `address` parameter.

NetStream.time

```
ns.time
```

Read-only; indicates the number of seconds the stream has been publishing. This is a good indication of whether data is flowing from the source that has been set in a call to the `NetStream.attach()` method.

Availability
Flash Media Server 3

ProxyStream class

Use the ProxyStream class to build large-scale applications that use DVR functionality. DVR functionality lets users pause live video and resume playback from the paused location. Users can rewind and play recorded sections of the video and seek forward to catch up again.

To scale applications, use server-side NetConnection objects to create a chain of servers. In such a scenario, all the servers run as local servers. To explain this scenario, the ingest servers at the top layer are called *origin servers*, servers in the middle layer are called *intermediate servers*, and the servers on the bottom layer (which serve subscribers) are called *edge servers*.

Note: *The terms "origin" and "edge" in this scenario do not refer to local and remote operation (see the* `Proxy` *section of the Vhost.xml configuration file). All servers in this scenario run in local mode.*

Only the server controlled by the publisher records streams. Servers further down the chain, closer to subscribers, do not record the stream and do not have the stream available for playback. If a recorded stream does not exist on a server, subscribers cannot access it. Use the ProxyStream class to pull segments of recorded streams from another server where the segments are available. Call `ProxyStream.proxyFrom()` to line up an intermediate server with an origin server. When a recorded stream is requested in the intermediate server, it automatically pulls the required segment down from the origin server. The segment is stored in the memory cache of the intermediate server just as segments are stored in the origin server. Every subscriber going through the intermediate server gets the data directly from the cache so that segments can be shared across multiple clients. Set up an edge server to pull data from an intermediate server, which pulls data from the origin server when necessary. It's a good idea to code your application so that when streams are idle they failover to a different source.

No disk usage is required in the intermediate and edge server. File segments that have been pulled from another server are stored in the memory cache. The server uses an LRU (Least Recently Used) scheme to maintain the cache. The server pushes older segments out of the cache when the memory reaches a value that you can configure. To save bandwidth and improve performance, the server can also save segments in the memory on disk. You can configure the location of the cache directory, and the maximum size of the cache.

Availability

Flash Media Server 3.5

Method summary

Method	Description
`ProxyStream.proxyFrom()`	Proxies a stream from one Adobe Media Server to another over a NetConnection.
`ProxyStream.stop()`	Stops proxying a stream.

Event handler summary

Event handler	Description
`ProxyStream.onStatus()`	Called every time a status change or error occurs in a ProxyStream object.

ProxyStream constructor

```
new ProxyStream(connection)
```

Creates an instance of the ProxyStream class.

Availability
Flash Media Server 3.5.

Parameters
`connection` A NetConnection object.

Returns
A ProxyStream object.

Example
The following example creates an instance of the ProxyStream class:

```
nc = new NetConnection();
nc.connect("rtmp://amsexample.adobe.com/testapp");
nc.onStatus(info){
    if(info.code == "NetConnection.Connect.Success"){
        ps = new ProxyStream(nc);
        // Use ps.onStatus to check status
    }
};
```

ProxyStream.onStatus()

```
ps.onStatus = function(infoObject){})
```

Called every time a status change or error occurs in a ProxyStream object.

Availability
Flash Media Server 3.5

Parameters
`infoObject` An Object with `code` and `level` properties that provide information about the status of a ProxyStream object. Both properties are strings.

Code property	Level property	Description
`ProxyStream.Proxy.Start`	`status`	Successfully published the source stream.
`ProxyStream.Proxy.Stop`	`status`	Successfully stopped the source stream.
`ProxyStream.Proxy.BadName`	`error`	The publish attempt failed because the local name was invalid or exists.

Example

```
nc = new NetConnection;
nc.onStatus = function(info) {
    if (info.code == "NetConnection.Connect.Success"){
    // Create a proxy to the remote stream "remoteStream"
    // This stream is published locally as "localStream"
    ps = new ProxyStream(nc);
    ps.onStatus =  function(info) {
        if (info.code == "ProxyStream.Proxy.Start") {
           // The local stream was published.
        }
        else if (info.code == "ProxyStream.Proxy.Stop") {
           // The local stream was stopped.
        }
        else if (info.code == "ProxyStream.Proxy.BadName") {
           // The publish failed because the local name was invalid
           // or existed.
        }
    };
    ps.proxyFrom("localStream", "remoteStream");
  }
};
nc.connect("rtmp://origin.mydvr.com/dvr");
```

The following example uses two origin connections, `primaryURI` and `backupURI` to demonstrate failover:

```
var primaryUri = "rtmp://primary/app";
var backupUri = "rtmp://backup//app";
function createNetConn(uri){
  _root.netConn = new NetConnection();
  _root.netConn.onStatus = function(info) {
    if (info.code == "NetConnection.Connect.Success"){
       // Connection is good, create the ProxyStream
       createProxyStream();
    }
    else if (info.code == "NetConnection.Proxy.NotResponding" ||
         info.code == "NetConnection.Connect.Closed") {
      // The proxy isn't responding to our requests, or the connection was closed,
      // so switch to an alternate origin...
      var nextUri;
      if (uri == primaryUri){
        nextUri = backupUri;
      }
      else {
        nextUri = primaryUri;
      }
      createNetConn(nextUri);
    }
  };
  _root.netConn.connect(uri);
}
function createProxyStream() {
  _root.ps = new ProxyStream(_root.netConn);
  _root.ps.onStatus =  function(info) {
     // Handle ProxyStream status notifications
  };
  _root.ps.proxyFrom("localStream", "remoteStream");
}
createNetConn(primaryUri);
```

ProxyStream.proxyFrom()

```
ps.proxyFrom(local, remote)
```

Proxies a stream from one Adobe Media Server to another over a NetConnection. The stream can be live, recorded, or both.

Availability
Flash Media Server 3.5

Parameters
`local` A string specifying a local stream name.

`remote` A string specifying a remote stream name.

ProxyStream.stop()

```
ps.stop()
```

Stops proxying a stream.

Availability
Flash Media Server 3.5

Parameters
None.

SharedObject class

The SharedObject class lets you store data on the server and share data between multiple client applications in real time. Shared objects can be temporary, or they can persist on the server after an application has closed; you can consider shared objects as real-time data transfer devices.

Note: *The following information explains the server-side SharedObject class. You can also create shared objects with the client-side SharedObject class.*

The following list describes common ways to use shared objects in Server-Side ActionScript:

1 Storing and sharing data on a server. A shared object can store data on the server for other clients to retrieve. For example, you can open a remote shared object, such as a phone list, that is persistent on the server. Whenever a client makes a change to the shared object, the revised data is available to all clients that are currently connected to the object or that connect to it later. If the object is also persistent locally and a client changes the data while not connected to the server, the changes are copied to the remote shared object the next time the client connects to the object.

2 Sharing data in real time. A shared object can share data among multiple clients in real time. For example, you can open a remote shared object that stores real-time data that is visible to all clients connected to the object, such as a list of users connected to a chat room. When a user enters or leaves the chat room, the object is updated and all clients that are connected to the object see the revised list of chat-room users.

   It is important to understand the following information about using shared objects in Server-Side ActionScript:

   - The Server-Side ActionScript method `SharedObject.get()` creates remote shared objects; there is no server-side method that creates local shared objects. Local shared objects are stored in memory, unless they're persistent, in which case they are stored in .sol files.
   - Remote shared objects that are stored on the server have the file extension .fso and are stored in a subdirectory of the application that created them. Remote shared objects on the client have the file extension .sor and are also stored in a subdirectory of the application that created them.
   - Server-side shared objects can be *nonpersistent* (that is, they exist for the duration of an application instance) or *persistent* (that is, they are stored on the server after an application closes).
   - To create a persistent shared object, set the `persistence` parameter of the `SharedObject.get()` method to `true`. Persistent shared objects let you maintain an application's state.

3 Every remote shared object is identified by a unique name and contains a list of name-value pairs, called *properties,* like any other ActionScript object. A name must be a unique string and a value can be any ActionScript data type.

   Note: *Unlike client-side shared objects, server-side shared objects do not have a* `data` *property.*

   - To get the value of a server-side shared object property, call `SharedObject.getProperty()`. To set the value of a server-side shared object property, call `SharedObject.setProperty()`.
   - To clear a shared object, call the `SharedObject.clear()` method; to delete multiple shared objects, call the `application.clearSharedObjects()` method.

- Server-side shared objects can be owned by the current application instance or by another application instance. The other application instance can be on the same server or on a different server. References to shared objects that are owned by a different application instance are called *proxied shared objects*.

If you write a server-side script that modifies multiple properties, you can prevent other clients from modifying the object during the update by calling the `SharedObject.lock()` method before updating the object. Then you can call `SharedObject.unlock()` to commit the changes and allow other changes to be made. Call `SharedObject.mark()` to deliver `change` events in groups within the `lock()` and `unlock()` methods.

When you get a reference to a proxied shared object, any changes made to the object are sent to the instance that owns the object. The success or failure of any changes is sent by using the `SharedObject.onSync()` event handler, if it is defined.

The `SharedObject.lock()` and `SharedObject.unlock()` methods cannot lock or unlock proxied shared objects.

Availability

Flash Communication Server 1

Property summary

Property	Description
`SharedObject.autoCommit`	A boolean value indicating whether the server periodically stores all persistent shared objects (`true`) or not (`false`).
`SharedObject.isDirty`	Read-only; a boolean value indicating whether the persistent shared object has been modified since the last time it was stored (`true`) or not (`false`).
`SharedObject.name`	Read-only; the name of a shared object.
`SharedObject.resyncDepth`	An integer that indicates when the deleted values of a shared object should be permanently deleted.
`SharedObject.version`	Read-only; the current version number of a shared object.

Method summary

Method	Description
`SharedObject.clear()`	Deletes all the properties of a single shared object and sends a clear event to all clients that subscribe to a persistent shared object.
`SharedObject.close()`	Detaches a reference from a shared object.
`SharedObject.commit()`	Static; stores either a specific persistent shared object instance or all persistent shared object instances with an `isDirty` property whose value is `true`.
`SharedObject.flush()`	Saves the current state of a persistent shared object.
`SharedObject.get()`	Static; creates a shared object or returns a reference to an existing shared object.
`SharedObject.getProperty()`	Retrieves the value of a named property in a shared object.
`SharedObject.getPropertyNames()`	Enumerates all the property names for a given shared object.
`SharedObject.lock()`	Locks a shared object.
`SharedObject.mark()`	Delivers all change events to a subscribing client as a single message.
`SharedObject.purge()`	Causes the server to remove all deleted properties that are older than the specified version.
`SharedObject.send()`	Executes a method in a client-side script.

Method	Description
SharedObject.setProperty()	Updates the value of a property in a shared object.
SharedObject.size()	Returns the total number of valid properties in a shared object.
SharedObject.unlock()	Allows other clients to update the shared object.

Event handler summary

Event handler	Description
SharedObject.handlerName()	An event handler invoked when a shared object receives a message with the same name from the client-side SharedObject.send() method.
SharedObject.onStatus()	Invoked when errors, warnings, and status messages associated with either a local instance of a shared object or a persistent shared object occur.
SharedObject.onSync()	Invoked when a shared object changes.

SharedObject.autoCommit

```
so.autoCommit
```

A boolean value indicating whether the server periodically stores all persistent shared objects (`true`) or not (`false`). If `autoCommit` is `false`, the application must call `SharedObject.commit()` to save the shared object; otherwise, the data is lost.

This property is `true` by default. To override the default, specify the initial state by using the following configuration key in the Application.xml file, as shown in the following example:

```
<SharedObjManager>
    <AutoCommit>false</AutoCommit>
</SharedObjManager>
```

Availability

Flash Media Server 2

SharedObject.clear()

```
so.clear()
```

Deletes all the properties of a single shared object and sends a `clear` event to all clients that subscribe to a persistent shared object. The persistent data object is also removed from a persistent shared object.

Availability

Flash Communication Server 1

Returns

Returns `true` if successful; otherwise, `false`.

See also

application.clearSharedObjects()

SharedObject.close()

```
so.close()
```

Detaches a reference from a shared object. A call to the `SharedObject.get()` method returns a reference to a shared object instance. The reference is valid until the variable that holds the reference is no longer in use and the script is garbage collected. To destroy a reference immediately, you can call `SharedObject.close()`. You can use `SharedObject.close()` when you no longer want to proxy a shared object.

Availability

Flash Communication Server 1

Example

In the following example, `so` is attached as a reference to shared object `foo`. When you call `so.close()`, you detach the reference `so` from the shared object `foo`.

```
so = SharedObject.get("foo");
    // Insert code here.
so.close();
```

See also

`SharedObject.get()`

SharedObject.commit()

```
so.commit([name])
```

Static; stores either a specific persistent shared object instance or all persistent shared object instances with an `isDirty` property whose value is `true`. Use this method if the `SharedObject.autoCommit` property is `false` and you need to manage when a shared object is stored locally.

Availability

Flash Media Server 2

Parameters

`name` A string indicating the name of the persistent shared object instance to store. If no name is specified, or if an empty string is passed, all persistent shared objects are stored. This parameter is optional.

Returns

A boolean value indicating success (`true`) or failure (`false`).

Example

The following code commits all dirty shared objects to local storage when the application stops:

```
application.onAppStop = function (info){
    // Insert code here.
    SharedObject.commit();
}
```

SharedObject.flush()

```
so.flush()
```

Saves the current state of a persistent shared object. Invokes the `SharedObject.onStatus()` handler and passes it an object that contains information about the success or failure of the call.

Availability
Flash Communication Server 1

Returns
A boolean value of `true` if successful; otherwise, `false`.

Example
The following example places a reference to the shared object `foo` in the variable `so`. It then locks the shared object instance so that no one can make any changes to it and saves the shared object by calling `so.flush()`. After the shared object is saved, it is unlocked so that further changes can be made.

```
var so = SharedObject.get("foo", true);
so.lock();
// Insert code here that operates on the shared object.
so.flush();
so.unlock();
```

SharedObject.get()

```
SharedObject.get(name, persistence [, netConnection])
```

Static; creates a shared object or returns a reference to an existing shared object. To perform any operation on a shared object, the server-side script must get a reference to the shared object by using the `SharedObject.get()` method. If the requested object is not found, a new instance is created.

Availability
Flash Communication Server 1

Parameters
`name` Name of the shared object instance to return.

`persistence` A boolean value: `true` for a persistent shared object; `false` for a nonpersistent shared object. If no value is specified, the default value is `false`.

`netConnection` A NetConnection object that represents a connection to an application instance. You can pass this parameter to get a reference to a shared object on another server or a shared object that is owned by another application instance. All update notifications for the shared object specified by the `name` parameter are proxied to this instance, and the remote instance notifies the local instance when a persistent shared object changes. The NetConnection object that is used as the `netConnection` parameter does not need to be connected when you call `SharedObject.get()`. The server connects to the remote shared object when the NetConnection state changes to connected. This parameter is optional.

Returns
A reference to an instance of the SharedObject class.

Details

There are two types of shared objects, persistent and nonpersistent, and they have separate namespaces. This means that a persistent and a nonpersistent shared object can have the same name and exist as two distinct shared objects. Shared objects are scoped to the namespace of the application instance and are identified by a string. The shared object names should conform to the URI specification.

You can also call `SharedObject.get()` to get a reference to a shared object that is in a namespace of another application instance. This instance can be on the same server or on a different server and is called a *proxied shared object*. To get a reference to a shared object from another instance, create a NetConnection object and use the `NetConnection.connect()` method to connect to the application instance that owns the shared object. Pass the NetConnection object as the `netConnection` parameter of the `SharedObject.get()` method. The server-side script must get a reference to a proxied shared object before there is a request for the shared object from any client. To do this, call `SharedObject.get()` in the `application.onAppStart()` handler.

If you call `SharedObject.get()` with a `netConnection` parameter and the local application instance already has a shared object with the same name, the shared object is converted to a proxied shared object. All shared object messages for clients that are connected to a proxied shared object are sent to the master instance.

If the connection state of the NetConnection object that was used as the `netConnection` parameter changes state from connected to disconnected, the proxied shared object is set to idle and any messages received from subscribers are discarded. The `NetConnection.onStatus()` handler is called when a connection is lost. You can then reestablish a connection to the remote instance and call `SharedObject.get()`, which changes the state of the proxied shared object from idle to connected.

If you call `SharedObject.get()` with a new NetConnection object on a proxied shared object that is already connected, and if the URI of the new NetConnection object doesn't match the current NetConnection object, the proxied shared object unsubscribes from the previous shared object, sends a `clear` event to all subscribers, and subscribes to the new shared object instance. When a subscribe operation to a proxied shared object is successful, all subscribers are reinitialized to the new state. This process lets you migrate a shared object from one application instance to another without disconnecting the clients.

Updates received by proxied shared objects from subscribers are checked to see if the update can be rejected based on the current state of the proxied shared object version and the version of the subscriber. If the change can be rejected, the proxied shared object doesn't forward the message to the remote instance; the reject message is sent to the subscriber.

The corresponding client-side ActionScript method is `SharedObject.getRemote()`.

Example

The following example creates a shared object named `foo` in the function `onProcessCmd()`. The function is passed a parameter, `cmd`, that is assigned to a property in the shared object.

```
function onProcessCmd(cmd){
    // Insert code here.
    var shObj = SharedObject.get("foo", true);
    propName = cmd.name;
    shObj.getProperty (propName, cmd.newAddress);
}
```

The following example uses a proxied shared object. A proxied shared object resides on a server or in an application instance (called *master*) that is different from the server or application instance that the client connects to (called *proxy*). When the client connects to the proxy and gets a remote shared object, the proxy connects to the master and gives the client a reference to this shared object. The following code is in the main.asc file:

```
application.appStart = function() {
    nc = new NetConnection();
    nc.connect("rtmp://" + master_server + "/" + master_instance);
    proxySO = SharedObject.get("myProxy",true,nc);
        // Now, whenever the client asks for a persistent
        // shared object called myProxy, it receives themyProxy
        // shared object from master_server/master_instance.
};
```

SharedObject.getProperty()

```
so.getProperty(name)
```

Retrieves the value of a named property in a shared object. The returned value is a copy associated with the property, and any changes made to the returned value do not update the shared object. To update a property, use the SharedObject.setProperty() method.

Availability

Flash Communication Server 1

Parameters

`name` A string indicating the name of a property in a shared object.

Returns

The value of a SharedObject property. If the property doesn't exist, returns `null`.

Example

The following example gets the value of the `name` property on the `user` shared object and assigns it to the `firstName` variable:

```
firstName = user.getProperty("name");
```

See also

`SharedObject.setProperty()`

SharedObject.getPropertyNames()

```
so.getPropertyNames()
```

Enumerates all the property names for a given shared object.

Availability

Flash Communication Server 1

Returns

An array of strings that contain all the property names of a shared object.

Example

The following example calls `getPropertyNames()` on the `myInfo` shared object and places the names in the `names` variable. It then enumerates those property names in a `for` loop.

```
myInfo = SharedObject.get("foo");
var addr = myInfo.getProperty("address");
myInfo.setProperty("city", "San Francisco");
var names = myInfo.getPropertyNames();
for (x in names){
    var propVal = myInfo.getProperty(names[x]);
    trace("Value of property " + names[x] + " = " + propVal);
}
```

SharedObject.handlerName()

```
so.handlerName = function([p1,..., pN]){}
```

An event handler invoked when a shared object receives a message with the same name from the client-side `SharedObject.send()` method. You must define a Function object and assign it to the event handler.

The `this` keyword used in the body of the function is set to the shared object instance returned by `SharedObject.get()`.

If you don't want the server to receive a particular message, do not define this handler.

Availability
Flash Communication Server 1

Parameters
`p1, ..., pN` Optional parameters passed to the handler method if the message contains user-defined parameters. These parameters are the user-defined objects that are passed to the `SharedObject.send()` method.

Returns
Any return value is ignored by the server.

Example
The following example defines an event handler called `traceArgs`:

```
var so = SharedObject.get("userList", false);
so.traceArgs = function(msg1, msg2){
    trace(msg1 + " : " + msg2);
};
```

SharedObject.isDirty

```
so.isDirty
```

Read-only; a boolean value indicating whether a persistent shared object has been modified since the last time it was stored (`true`) or not (`false`). The `SharedObject.commit()` method stores shared objects with an `isDirty` property that is `true`.

This property is always `false` for nonpersistent shared objects.

Availability
Flash Media Server 2

Example
The following example saves the `so` shared object if it has been changed:

```
var so = SharedObject.get("foo", true);
if (so.isDirty){
    SharedObject.commit(so.name);
}
```

SharedObject.lock()

```
so.lock()
```

Locks a shared object. This method gives the server-side script exclusive access to the shared object; when the `SharedObject.unlock()` method is called, all changes are batched and one update message is sent through the `SharedObject.onSync()` handler to all the clients that subscribe to this shared object. If you nest the `SharedObject.lock()` and `SharedObject.unlock()` methods, make sure that there is an `unlock()` method for every `lock()` method; otherwise, clients are blocked from accessing the shared object.

You cannot use the `SharedObject.lock()` method on proxied shared objects.

Availability
Flash Communication Server 1

Returns
An integer indicating the lock count: 0 or greater indicates success; -1 indicates failure. For proxied shared objects, always returns -1.

Example
The following example locks the `so` shared object, executes the code that is to be inserted, and then unlocks the object:

```
var so = SharedObject.get("foo");
so.lock();
// Insert code here that operates on the shared object.
so.unlock();
```

SharedObject.mark()

```
so.mark(handlerName, p1, ..., pN)
```

Delivers all `change` events to a subscribing client as a single message.

In a server-side script, you can call the `SharedObject.setProperty()` method to update multiple shared object properties between a call to the `lock()` and `unlock()` methods. All subscribing clients receive a `change` event notification through the `SharedObject.onSync()` handler. However, because the server may collapse multiple messages to optimize bandwidth, the `change` event notifications may not be sent in the same order as they were in the code.

Use the `mark()` method to execute code after all the properties in a set have been updated. You can call the `handlerName` parameter passed to the `mark()` method, knowing that all property changes before the `mark()` call have been updated.

Availability
Flash Media Server 2

Parameters

`handlerName` Calls the specified handler on the client-side SharedObject instance. For example, if the `handlerName` parameter is `onChange`, the client invokes the `SharedObject.onChange()` handler with all the `p1, ...,pN` parameters.

Note: *Do not use a built-in method name for a handler name. For example, if the handler name is `close`, the subscribing stream will be closed.*

`p1, ..., pN` Parameters of any ActionScript type, including references to other ActionScript objects. Parameters are passed to `handlerName` when it is executed on the client.

Returns

A boolean value. Returns `true` if the message can be dispatched to the client; otherwise, `false`.

Example

The following example calls the `mark()` method twice to group two sets of shared object property updates for clients:

```
var myShared = SharedObject.get("foo", true);

myShared.lock();
myShared.setProperty("name", "Stephen");
myShared.setProperty("address", "Xyz lane");
myShared.setProperty("city", "SF");
myShared.mark("onAdrChange", "name");
myShared.setProperty("account", 12345);
myShared.mark("onActChange");
myShared.unlock();
```

The following example shows the receiving client-side script:

```
connection = new NetConnection();
connection.connect("rtmp://flashmediaserver/someApp");
var x = SharedObject.get( "foo", connection.uri, true);
x.connect(connection);
x.onAdrChange = function(str) {
    // Shared object has been updated,
    // can look at the "name", "address" and "city" now.
}

x.onActChange = function(str) {
    // Shared object has been updated,
    // can look at the "account" property now,
}
```

SharedObject.name

`so.name`

Read-only; the name of a shared object.

Availability

Flash Communication Server 1

SharedObject.onStatus()

```
so.onStatus = function(info) {}
```

Invoked when errors, warnings, and status messages associated with either a local instance of a shared object or a persistent shared object occur.

Availability

Flash Communication Server 1

Parameters

`info` An information object.

Example

The following client-side code defines an anonymous function that just traces the `level` and `code` properties of the specified shared object:

```
so = SharedObject.get("foo", true);
so.onStatus = function(infoObj){
    //Handle status messages passed in infoObj.
    trace(infoObj.level + "; " + infoObj.code);
};
```

SharedObject.onSync()

```
so.onSync = function(list){}
```

Invoked when a shared object changes. Use the `onSync()` handler to define a function that handles changes made to a shared object by subscribers.

For proxied shared objects, defines the function to get the status of changes made by the server and other subscribers.

Note: *You cannot define the `onSync()` handler on the `prototype` property of the SharedObject class in Server-Side ActionScript.*

Availability

Flash Communication Server 1

Parameters

`list` An array of objects that contain information about the properties of a shared object that have changed since the last time the `onSync()` handler was called. The notifications for proxied shared objects are different from the notifications for shared objects that are owned by the local application instance. The following table describes the codes for local shared objects:

Local code	Meaning
`change`	A property was changed by a subscriber.
`delete`	A property was deleted by a subscriber.
`name`	The name of a property that has changed or been deleted.
`oldValue`	The old value of a property. This is true for both `change` and `delete` messages; on the client, `oldValue` is not set for `delete`.

Note: *Changing or deleting a property on the server side by using the* `SharedObject.setProperty()` *method always succeeds, so there is no notification of these changes.*

The following table describes the codes for local shared objects:

Proxied code	Meaning
`success`	A server change of the shared object was accepted.
`reject`	A server change of the shared object was rejected. The value on the remote instance was not changed.
`change`	A property was changed by another subscriber.
`delete`	A property was deleted. This notification can occur when a server deletes a shared object or if another subscriber deletes a property.
`clear`	All the properties of a shared object are deleted. This can happen when the server's shared object is out of sync with the master shared object or when the persistent shared object migrates from one instance to another. This event is typically followed by a `change` message to restore all of the server's shared object properties.
`name`	The name of a property that has changed or been deleted.
`oldValue`	The old value of the property. This is valid only for the reject, change, and delete codes.

Note: *The* `SharedObject.onSync()` *handler is invoked when a shared object has been successfully synchronized with the server. If there is no change in the shared object, the list object may be empty.*

Example

The following example creates a function that is invoked whenever a property of the shared object `so` changes:

```
// Create a new NetConnection object.
nc = new NetConnection();
nc.connect("rtmp://server1.xyx.com/myApp");
// Create the shared object.
so = SharedObject.get("MasterUserList", true, nc);
// The list parameter is an array of objects containing information
// about successfully or unsuccessfully changed properties
// from the last time onSync() was called.
so.onSync = function(list) {
    for (var i = 0; i < list.length; i++) {
        switch (list[i].code ) {
            case "success":
                trace ("success");
                break;
            case "change":
                trace ("change");
                break;
            case "reject":
                trace ("reject");
                break;
            case "delete":
                trace ("delete");
                break;
            case "clear":
                trace ("clear");
                break;
        }
    }
};
```

SharedObject.purge()

`so.purge(version)`

Causes the server to remove all deleted properties that are older than the specified version. Although you can also accomplish this task by setting the `SharedObject.resyncDepth` property, the `purge()` method gives the script more control over which properties to delete.

Availability
Flash Communication Server 1

Parameters
`version` A number indicating the version. All deleted data that is older than this version is removed.

Returns
A boolean value.

Example
The following example deletes all the properties of the `so` shared object that are older than the value of `so.version - 3`:

```
var so = SharedObject.get("foo", true);
so.lock();
so.purge(so.version - 3);
so.unlock();
```

SharedObject.resyncDepth

`so.resyncDepth`

An integer that indicates when the deleted properties of a shared object should be permanently deleted. You can use this property in a server-side script to resynchronize shared objects and to control when shared objects are deleted. The default value is infinity.

If the current revision number of the shared object minus the revision number of the deleted property is greater than the value of `SharedObject.resyncDepth`, the property is deleted. Also, if a client connecting to this shared object has a client revision that, when added to the value of `SharedObject.resyncDepth`, is less than the value of the current revision on the server, all the current elements of the client shared object are deleted, the valid properties are sent to the client, and the client receives a "clear" message.

This method is useful when you add and delete many properties and you don't want to send too many messages to the client. Suppose that a client is connected to a shared object that has 12 properties and then disconnects. After that client disconnects, other clients that are connected to the shared object delete 20 properties and add 10 properties. When the client reconnects, it could, for example, receive a delete message for the 10 properties it previously had and then a change message on two properties. You can use `SharedObject.resyncDepth` property to send a "clear" message, followed by a change message for two properties, which saves the client from receiving 10 delete messages.

Availability
Flash Communication Server 1

Example
The following example resynchronizes the shared object `so` if the revision number difference is greater than 10:

```
so = SharedObject.get("foo");
so.resyncDepth = 10;
```

SharedObject.send()

```
so.send(methodName, [p1, ..., pN])
```

Executes a method in a client-side script. You can use `SharedObject.send()` to asynchronously execute a method on all the Flash clients subscribing to a shared object. The server does not receive any notification from the client on the success, failure, or return value in response to this message.

Availability

Flash Communication Server 1

Parameters

`methodName` A string indicating the name of a method on a client-side shared object. For example, if you specify "`doSomething`", the client must invoke the `SharedObject.doSomething()` method, with all the `p1, ..., pN` parameters.

`p1, ..., pN` Parameters of any type, including references to other objects. These parameters are passed to the specified `methodName` on the client.

Returns

A boolean value of `true` if the message was sent to the client; otherwise, `false`.

Example

The following example calls the `SharedObject.send()` method to invoke the `doSomething()` method on the client and passes the string `"This is a test"`:

```
var so = SharedObject.get("foo", true);
so.send("doSomething", "This is a test");
```

The following example is the client-side ActionScript code that defines the `doSomething()` method:

```
nc = new NetConnection();
nc.connect("rtmp://www.adobe.com/someApp");
var so = SharedObject.getRemote("foo", nc.uri, true);
so.connect(nc);
so.doSomething = function(str) {
// Process the str object.
};
```

SharedObject.setProperty()

```
so.setProperty(name, value)
```

Updates the value of a property in a shared object.

The `name` parameter on the server side is the same as an attribute of the `data` property on the client side. For example, the following two lines of code are equivalent; the first line is Server-Side ActionScript and the second is client-side ActionScript:

```
so.setProperty(nameVal, "foo");
clientSO.data[nameVal] = "foo";
```

A shared object property can be modified by a client between successive calls to `SharedObject.getProperty()` and `SharedObject.setProperty()`. If you want to preserve transactional integrity, call the `SharedObject.lock()` method before modifying the shared object; be sure to call `SharedObject.unlock()` when you finish making modifications. If you call `SharedObject.setProperty()` without first calling `SharedObject.lock()`, the change is made to the shared object, and all object subscribers are notified before `SharedObject.setProperty()` returns. If you call `SharedObject.lock()` before you call `SharedObject.setProperty()`, all changes are batched and sent when the `SharedObject.unlock()` method is called. The `SharedObject.onSync()` handler on the client side is invoked when the local copy of the shared object is updated.

Note: *If only one source (whether client or server) is updating a shared object in a server-side script, you don't need to use the `lock()` or `unlock()` method or the `onSync()` handler.*

Availability
Flash Communication Server 1

Parameters
`name` The name of the property in the shared object.

`value` An ActionScript object associated with the property, or `null` to delete the property.

Example
The following example uses the `SharedObject.setProperty()` method to create the `city` property with the value `San Francisco`. It then enumerates all the property values in a `for` loop and calls `trace()` to display the values.

```
myInfo = SharedObject.get("foo");
var addr = myInfo.getProperty("address");
myInfo.setProperty("city", "San Francisco");
var names = sharedInfo.getPropertyNames();
for (x in names){
    var propVal = sharedInfo.getProperty(names[x]);
    trace("Value of property " + names[x] + " = " + propVal);
}
```

See also
`SharedObject.getProperty()`

SharedObject.size()

`so.size()`

Returns the total number of valid properties in a shared object.

Availability
Flash Communication Server 1

Returns
An integer indicating the number of properties.

Example
The following example gets the number of properties of a shared object and assigns that number to the variable `len`:

```
var so = SharedObject.get("foo", true);
var soLength = so.size();
```

SharedObject.unlock()

```
so.unlock()
```

Allows other clients to update the shared object. A call to this method also causes the server to commit all changes made after the `SharedObject.lock()` method is called and sends an update message to all clients.

You cannot call the `SharedObject.unlock()` method on proxied shared objects.

Availability

Flash Communication Server 1

Returns

An integer indicating the lock count: 0 or greater if successful; -1 otherwise. For proxied shared objects, this method always returns -1.

Example

The following example unlocks a shared object:

```
var so = SharedObject.get("foo", true);
so.lock();
// Insert code to manipulate the shared object.
so.unlock();
```

See also

`SharedObject.lock()`

SharedObject.version

```
so.version
```

Read-only; the current version number of the shared object. Calls to the `SharedObject.setProperty()` method on either the client or the server increment the value of the `version` property.

Availability

Flash Communication Server 1

SHA256 class

The Server-Side ActionScript SHA256 class is a symmetric port of the Flex® SDK mx.util.SHA256 utility class. Use this class to generate a digest, or signature, for binary data. A receiver of the data can compute its own digest and compare that to the original digest value to ensure that the binary data has not been tampered with.

Method	Description
`SHA256.computeDigest()`	Uses the SHA-256 hash algorithm to compute the digest of a message.

SHA256.computeDigest()

`SHA256.computeDigest(bytes)`

Uses the SHA-256 hash algorithm to compute the digest of a message.

The function throws a JavaScript Error if the `bytes` argument is null or not a ByteArray.

Availability
Flash Media Server 4

Parameters
`bytes` A ByteArray containing the message.

Returns
A String that is a 64 character hexadecimal representation of the digest.

SOAPCall class

Availability
Flash Media Server 2

The SOAPCall class is the object type that is returned from all web service calls. These objects are typically constructed automatically when a Web Service Definition Language (WSDL) is parsed and a stub is generated.

Property summary

Property	Description
`SOAPCall.request`	An XML object that represents the current SOAP (Simple Object Access Protocol) request.
`SOAPCall.response`	An XML object that represents the most recent SOAP response.

Event handler summary

Event handler	Description
`SOAPCall.onFault()`	Invoked when a method has failed and returned an error.
`SOAPCall.onResult()`	Invoked when a method has successfully invoked and returned.

SOAPCall.onFault()

`SOAPCall.onFault(fault)`

Invoked when a method has failed and returned an error.

Availability
Flash Media Server 2

Parameters
`fault` The `fault` parameter is an object version of an XML SOAP Fault (see SOAPCall class).

SOAPCall.onResult()

```
mySOAPCall.onResult(result){}
```

Invoked when a method has been successfully invoked and returned.

Availability
Flash Media Server 2

Parameters
`result` The decoded ActionScript object returned by the operation (if any). To get the raw XML returned instead of the decoded result, access the `SOAPCall.response` property.

SOAPCall.request

```
mySOAPCall.request
```

An XML object that represents the current Simple Object Access Protocol (SOAP) request.

Availability
Flash Media Server 2

SOAPCall.response

```
mySOAPCall.response
```

An XML object that represents the most recent SOAP response.

Availability
Flash Media Server 2

SOAPFault class

The SOAPFault class is the object type of the error object returned to the WebService.onFault() and `SOAPCall.onFault()` functions. This object is returned as the result of a failure and is an ActionScript mapping of the SOAP Fault XML type.

Availability
Flash Media Server 2

Property summary

Property	Description
`SOAPFault.detail`	A string indicating the application-specific information associated with the error, such as a stack trace or other information returned by the web service engine.
`SOAPFault.faultactor`	A string indicating the source of the fault.
`SOAPFault.faultcode`	A string indicating the short, standard qualified name describing the error.
`SOAPFault.faultstring`	A string indicating the human-readable description of the error.

SOAPFault.detail

```
mySOAPFault.detail
```

A string indicating the application-specific information associated with the error, such as a stack trace or other information returned by the web service engine.

Availability
Flash Media Server 2

SOAPFault.faultactor

```
mySOAPFault.faultactor
```

A string indicating the source of the fault. This property is optional if an intermediary is not involved.

Availability
Flash Media Server 2

SOAPFault.faultcode

```
mySOAPFault.faultcode
```

A string indicating the short, standard qualified name describing the error.

Availability
Flash Media Server 2

SOAPFault.faultstring

```
mySOAPFault.faultstring
```

A string indicating the human-readable description of the error.

Availability
Flash Media Server 2

Example
The following example shows the fault code in a text field if the WSDL fails to load:

```
// Load the WebServices class:
load("webservices/WebServices.asc");

// Prepare the WSDL location:
var wsdlURI = "http://www.flash-db.com/services/ws/companyInfo.wsdl";

// Instantiate the web service object by using the WSDL location:
stockService = new WebService(wsdlURI);

// Handle the WSDL parsing and web service instantiation event:
stockService.onLoad = function(wsdl){
    wsdlField.text = wsdl;
}

// If the wsdl fails to load, the onFault event is fired:
stockService.onFault = function(fault){
    wsdlField.text = fault.faultstring;
}
```

Stream class

The Stream class lets you manage or republish streams in an application. A Stream object is the server-side equivalent of the client-side NetStream object.

You can't attach audio or video sources to a Stream object; you can only play and manage existing streams. Use the Stream class to shuffle existing streams in a playlist, pull streams from other servers, and control access to streams. You can also record streams published by a client and record data streams such as log files.

A stream is a one-way connection between a client running Flash Player and a server running Adobe Media Server. A stream can also be a connection between two servers running Adobe Media Server. You can create a stream in Server-Side ActionScript by calling `Stream.get()`. A client can access multiple streams at the same time, and there can be hundreds or thousands of Stream objects active at the same time. You can record in FLV and F4V format.

Streams can contain ActionScript data. Call the `Stream.send()` method to add data to a stream. You can extract this data without waiting for a stream to play in real time, such as when you're creating a log file. You can also use it to add metadata to a stream.

Availability

Flash Communication Server 1

Property summary

Property	Description
`Stream.bufferTime`	Read-only; indicates how long to buffer messages before a stream is played, in seconds.
Stream.generateCCInfo	Read-write; indicates whether 608-708 captions are extracted from the H264 NALU and converted to `onCaptionInfo` AMF message.
`Stream.maxQueueDelay`	Read-only; the maximum time, in milliseconds, that the live queue can delay transmitting messages.
`Stream.maxQueueSize`	Read-only; the maximum size, in bytes, that the live queue can grow to before transmitting the messages it contains.
`Stream.name`	Read-only; contains a unique string associated with a live stream.

Property	Description
`Stream.publishQueryString`	Read-only; the query string specified in the stream path when the stream was published.
`Stream.receiveAudio`	A boolean value that controls whether the server receives audio (`true`) or not (`false`).
`Stream.receiveVideo`	A boolean value that controls whether the server receives video (`true`) or not (`false`).
Stream.time	Read-only; the number of seconds the stream has been playing. This value is the timestamp of the latest frame that flowed out of the stream.

Method summary

Method	Description
`Stream.clear()`	Deletes a recorded file from the server.
`Stream.destroy()`	Unlinks and cleans up a stream resource.
`Stream.flush()`	Flushes a stream.
`Stream.get()`	Static; returns a reference to a Stream object.
`Stream.getOnMetaData()`	Returns an object containing the metadata for the named stream or video file.
`Stream.length()`	Static; returns the length of a recorded stream in seconds.
`Stream.play()`	Controls the data source of a stream with an optional start time, duration, and reset flag to flush any previously playing stream.
`Stream.record()`	Records all the data passing through a Stream object and creates a file of the recorded stream.

Event handler summary

Event handler	Description
`Stream.onStatus()`	Invoked every time the status of a Stream object changes.

Stream.bufferTime

```
myStream.bufferTime
```

Read-only; indicates how long to buffer messages before a stream plays, in seconds. This property applies only when playing a stream from a remote server or when playing a recorded stream locally. Call `Stream.setBufferTime()` to set the `bufferTime` property.

A message is data that is sent back and forth between Adobe Media Server and Flash Player. The data is divided into small packets (messages), and each message has a type (audio, video, or data).

Availability

Flash Communication Server 1

Stream.clear()

```
myStream.clear()
```

Deletes a recorded FLV or F4V file from the server.

Availability
Flash Communication Server 1

Returns
A boolean value of `true` if the call succeeds; otherwise, `false`.

Example
The following example deletes a recorded stream called playlist.flv. Before the stream is deleted, the example defines an `onStatus()` handler that uses two information object error codes, `NetStream.Clear.Success` and `NetStream.Clear.Failed`, to send status messages to the application log file and the Live Log panel in the Administration Console.

```
s = Stream.get("playlist");
if (s){
    s.onStatus = function(info){
        if(info.code == "NetStream.Clear.Success"){
            trace("Stream cleared successfully.");
        }
        if(info.code == "NetStream.Clear.Failed"){
            trace("Failed to clear stream.");
        }
    };
    s.clear();
}
```

Stream.destroy()

```
Stream.destroy()
```

Unlinks and cleans up an instance of the Stream class. When a call to `Stream.destroy()` destroys a stream, the server stops any NetStream playing the stream and logs the code 440 in the Access log. Publishers are disconnected and recordings are stopped.

Instances of the Stream class are considered live streams. These streams are used as proxy streams, for server-to-sever streaming, direct client playback, and recordings. It can be difficult to determine whether you've properly released a live stream that has finished playing. Call `Stream.destroy()` to unlink and clean up a stream resource.

Availability
Flash Meda Server 3.5.4

Returns
A boolean value of `true` if the call succeeds.

A boolean value of `false` if the stream is not found. This case occurs only if the Stream reference has already been destroyed or if an error has occurred.

Example
The following example destroys a stream called streamA:

```
// Get a stream.
streamA = Stream.get("clientStream1");
// Destroy the stream.
Stream.destroy(streamA);
```

Stream.flush()

`myStream.flush()`

Flushes a stream. If the stream is used for recording, the `flush()` method writes the contents of the buffer associated with the stream to the recorded file.

It is highly recommended that you call `flush()` on a stream that contains only data. Synchronization problems can occur if you call the `flush()` method on a stream that contains data and either audio, video, or both.

Availability
Flash Media Server 2

Returns
A boolean value of `true` if the buffer was successfully flushed; otherwise, `false`.

Example
The following example flushes the `myStream` stream:

```
// Set up the server stream.
application.videoStream = Stream.get("aVideo");

if (application.videoStream){
    application.videoStream.record();
    application.videoStream.send("test", "hello world");
    application.videoStream.flush();
}
```

Stream.generateCCInfo

`myStream.generateCCInfo`

A Boolean value that indicates whether AMS extracts 608-708 captions from the H264 NALU and constructs corresponding `onCaptionInfo` AMF messages for use in HDS live and RTMP streams. This property overrides the `GenerateCCInfo` setting in the Application.xml file. The default value is `false`. A value of `true`, will enable closed captioning.

Note: *If your video streams are relayed from AMS server to AMS server, only the ingest server can generate captions. Setting `Stream.generateCCInfo` to `true` for non-ingest servers has no effect.*

For more information, see StreamManager.

Availability
Adobe Media Server 5.0.1

Stream.get()

`Stream.get(name)`

Static; returns a reference to a Stream object. If the requested object is not found, a new instance is created. After you call the `Stream.get()` method, you can call the `Stream.record()` and `Stream.play()` methods to publish and record streams.

You can publish and record streams in FLV, F4V, or F4F format. F4F format is for HTTP Dynamic Streaming.

Specify the format in the `name` parameter you pass to the `Stream.get()` method. To publish in FLV format, specify only the stream name, for example, `Stream.get("footballGame")`. To publish in F4V format, prefix the stream name with `mp4:`. You can optionally specify the file extension, for example, the following code is all legal:

```
Stream.get("mp4:footballGame.f4v")
Stream.get("mp4:footballGame.mp4")
Stream.get("mp4:footballGame")
```

F4V files behave differently than FLV files. To create a file with a file extension, you must specify a file extension. If you don't specify a file extension, the file created will not have a file extension.

To record one stream, create a stream with that format when you call `Stream.get()`. For example, if you want to record the stream "myHomeMovie.mp4", use code like the following:

```
s = Stream.get("mp4:streamName.mp4");
if(s){
    s.record();
    s.play("mp4:myHomeMovie.mp4");
}
```

To record in F4F format for HTTP Dynamic Streaming, do the following:

```
s = Stream.get("f4f:streamName.f4f");
if(s){
    s.record();
    s.play("mp4:myHomeMovie.mp4");
}
```

When you add streams to the end of an existing file to make a playlist, you might add streams with different settings and formats. If you record a file in FLV format, the server records the streams encoded with On2 VP6 and ignores streams encoded with H.264. If you record a file in F4V format, you can append any type of content to the stream, including FLV, MP3, MP4, F4V, and live streams.

Note: *To play or edit F4V files recorded by Adobe Media Server in other tools, use the Adobe Media Server F4V Post Processor tool. The tool is available at www.adobe.com/go/ams_tools.*

Availability

Flash Communication Server 1

Parameters

`name` A string specifying the name of a Stream object.

Returns

A Stream object if the call is successful; otherwise, `null`.

Examples

The following example publishes and records a video in F4V format. The stream contains 2 videos, one in FLV format, and one in MP4 format.

```
var s=Stream.get("mp4:streamName.f4v");
if(s) {
    s.record();
    s.play("sample",-2,-1);
    s.play("mp4:sample_mp4.mp4",-2,-1,false);
}
```

Stream.getOnMetaData()

`Stream.getOnMetaData(name)`

Static; returns an object containing the metadata for the named stream or video file. The object contains one property for each metadata item. The Flash Video Exporter utility (version 1.1 or later) embeds video duration, creation date, data rates, and other information into the video file.

Availability
Flash Media Server 2

Parameter
`name` A string indicating the name of a recorded stream, such as `"myVideo"`. For FLV files, pass the name without a file extension or prefix: `"myVideo"`. For F4V files, pass the name with the prefix `mp4:` `"mp4:myVideo"`. Append a file extension if the F4V file has a file extension.

Returns
An Object containing the metadata as properties.

Example
The following example lists the properties and values for the metadata for the recorded stream `myVideo.flv`:

```
var infoObject = Stream.getOnMetaData("myVideo");

trace("Metadata for myVideo.flv:");

for(i in infoObject){
    trace(i + " = " + infoObject[i]);
}
```

Stream.length()

`Stream.length(name[, virtualKey])`

Static; returns the length of a recorded file in seconds. If the requested file is not found, the return value is 0.

Availability
Flash Communication Server 1

Parameters
`name` A string indicating the name of a recorded stream. To get the length of an MP3 file, precede the name of the file with `mp3:` (for example, `"mp3:beethoven"`).

`virtualKey` A string indicating a key value. Starting with Adobe Media Server 2, stream names are not always unique. You can create multiple streams with the same name and place them in different physical directories. Then, use the `VirtualDirectory` section and `VirtualKeys` section of the Vhost.xml file to direct clients to the appropriate stream. The `Stream.length()` method is not associated with a client, but connects to a stream on the server. As a result, you may need to specify a virtual key to identify the correct stream. For more information about keys, see `Client.virtualKey`. This parameter is optional.

Returns
A number.

Example

The following example gets the length of the recorded stream file `myVideo` and assigns it to the variable `streamLen`:

```
function onProcessCmd(cmd){
    var streamLen = Stream.length("myVideo");
    trace("Length: " + streamLen + "\n");
}
```

The following example gets the length of the MP3 file `beethoven.mp3` and assigns it to the variable `streamLen`:

```
function onProcessCmd(cmd){
    var streamLen = Stream.length("mp3:beethoven");
    trace("Length: " + streamLen + "\n");
}
```

The following example gets the length of the MP4 file `beethoven.mp4` and assigns it to the variable `streamLen`:

```
function onProcessCmd(cmd){
    var streamLen = Stream.length("mp4:beethoven");
    trace("Length: " + streamLen + "\n");
}
```

Stream.liveEvent

`stream.liveEvent`

A string indicating the name of an HTTP Dynamic Streaming live event. For more information, see Using Adobe HTTP Dynamic Streaming .

Availability

Flash Media Server 3.8

Example

The following example sets the name of a live event to “myevent”:

```
application.onPublish = function(clientObj, streamObj){
    livestream = Stream.get("f4f:livestream");
    livestream.liveEvent = "myevent";
    livestream.record("record");
    livestream.play(streamObj.name, -1);
}
```

Stream.maxQueueDelay

`myStream.maxQueueDelay`

The maximum time, in milliseconds, that the live queue can delay transmitting messages.

Availability

Flash Media Server 3.5

Example

For an example, see the `Stream.publishQueryString` property.

See also

`Stream.maxQueueSize`, `Stream.publishQueryString`

Stream.maxQueueSize

```
myStream.maxQueueSize
```

The maximum size, in bytes, that the live queue can grow to before transmitting the messages it contains.

Availability

Flash Media Server 3.5

Example

For an example, see the `Stream.publishQueryString` property.

See also

`Stream.maxQueueDelay`, `Stream.publishQueryString`

Stream.name

```
myStream.name
```

Read-only; contains a unique string associated with a live stream. You can use this property as an index to find a stream within an application.

Availability

Flash Communication Server 1

Example

The following function takes a Stream object as a parameter and returns the name of the stream:

```
function getStreamName(myStream){
    return myStream.name;
}
```

Stream.onPlayStatus()

```
myStream.onPlayStatus = function(infoObject) {}
```

Called to provide information about playing a stream.

You can trigger actions when a Stream object has switched from one stream to another in a playlist (as indicated by the information object `NetStream.Play.Switch`). You can also trigger actions when a Stream object has played to the end (as indicated by the information object `NetStream.Play.Complete`). To respond to this event, you must create a function to process the information object sent by the server.

Availability

Flash Media Server 2; Flash Player 6.

Parameters

`infoObject` An object with `code` and `level` properties that provide information about the play status of a Stream object, as follows:

code property	level property	Meaning
`NetStream.Play.Complete`	`status`	Playback has completed.
`NetStream.Play.TransitionComplete`	`status`	The subscriber is switching to a new stream as a result of a successful `NetStream.play2()` call. For Flash Player 10 and later and Flash Media Server 3.5 and later. Adobe Media Server dispatches this event when a transition to a new stream occurs. Prior to this event, the server dispatches an `onStatus` event with a code of `NetStream.Play.Transition` to indicate that it processed the command to switch streams.
`NetStream.Play.Switch`	`status`	The subscriber is switching from one stream to another in a playlist.

Stream.onStatus()

```
myStream.onStatus = function([infoObject]) {}
```

Invoked every time the status of a Stream object changes. For example, if you play a file in a stream, `Stream.onStatus()` is invoked. Use `Stream.onStatus()` to check when play starts and ends, when recording starts, and so on.

Availability

Flash Communication Server 1

Parameters

`infoObject` An Object with `code` and `level` properties that contain information about a stream. This parameter is optional, but it is usually used. The Stream information object contains the following properties:

Property	Meaning
`code`	A string identifying the event that occurred.
`description`	Detailed information about the code. Not every information object includes this property.
`details`	The stream name.
`level`	A string indicating the severity of the event.

The following table describes the `code` and `level` property values:

Code property	Level property	Description
`NetStream.Clear.Failed`	`error`	A call to `application.clearStreams()` failed to delete a stream.
`NetStream.Clear.Success`	`status`	A call to `application.clearStreams()` successfully deleted a stream.
`NetStream.Failed`	`error`	An attempt to use a Stream method failed.
`NetStream.MulticastStream.Reset`	`status`	Dispatched when the low-level multicast stream indicates a reset point. This only happens on the subscribe-side of the stream.

Code property	Level property	Description
NetStream.Play.Failed	error	A call to Stream.play() failed. In Server-Side ActionScript, you can use the NetStream class to publish a stream to a multicast group. You cannot use the NetStream class to play a stream being published into a group.
NetStream.Play.InsufficientBW	warning	Data is playing behind the normal speed.
NetStream.Play.Start	status	Play was started.
NetStream.Play.StreamNotFound	error	An attempt was made to play a stream that does not exist.
NetStream.Play.Stop	status	Play was stopped.
NetStream.Play.Reset	status	A playlist was reset.
NetStream.Play.PublishNotify	status	The initial publish operation to a stream was successful. This message is sent to all subscribers.
NetStream.Play.UnpublishNotify	status	An unpublish operation from a stream was successful. This message is sent to all subscribers.
NetStream.Publish.BadName	error	An attempt was made to publish a stream that is already being published by someone else.
NetStream.Publish.Start	status	Publishing was started.
NetStream.Record.Failed	error	An attempt to record a stream failed.
NetStream.Record.NoAccess	error	An attempt was made to record a read-only stream.
NetStream.Record.Start	status	Recording was started.
NetStream.Record.Stop	status	Recording was stopped.
NetStream.Unpublish.Success	status	A stream has stopped publishing.

Example

The following server-side code attempts to delete a given stream and traces the resulting return code:

```
Client.prototype.delStream = function(streamName){
    trace("*** deleting stream: " + streamName);
    s = Stream.get("streamName");
    if (s) {
        s.onStatus = function(info){
            if (info.code == "NetStream.Clear.Success"){
                trace("*** Stream " + streamName + "deleted.");
            }
            if (info.code == "NetStream.Clear.Failure"){
                trace("*** Failure to delete stream " + streamName);
            }
        };
        s.clear();
    }
}
```

Stream.play()

```
myStream.play(streamName, [startTime, length, reset, remoteConnection, virtualKey])
```

Controls the data source of a stream with an optional start time, duration, and reset flag to flush any previously playing stream. Call `play()` to do the following:

- Chain streams between servers.
- Create a hub to switch between live streams and recorded streams.
- Combine streams into a recorded stream.

You can combine multiple streams to create a playlist for clients. The `Stream.play()` method behaves differently from the `NetStream.play()` method on the client side. A server-side call to `Stream.play()` is similar to a client-side call to `NetStream.publish()`; it controls the source of data coming into a stream. When you call `Stream.play()` on the server, the server becomes the publisher. Because the server has higher priority than the client, the client is forced to unpublish from the stream if the server calls a `play()` method on the same stream.

If any recorded streams are included in a server playlist, you cannot play the server playlist stream as a live stream.

Note: *A stream that plays from a remote server by means of the NetConnection object is considered a live stream.*

You do not need to wait for `"NetStatus.Connection.Success"` when connecting to another Adobe Media Server from Server-Side Actionscript. The server waits for the connection to complete before it attempts to use the connection for `Stream.play()`. However, you may want to monitor the NetStatusEvent so that you can handle a failed connection.

To delete a Stream object, use the `delete` operator to mark the stream for deletion. The script engine deletes the object during its garbage collection routine.

```
// Initialize the Stream object.
s = stream.get("foo");
// Play the stream.
s.play("name", p1, ... pN);
// Stop the stream.
s.play(false);
// Mark the Stream object for deletion during server garbage routine.
delete s;
```

Availability

Flash Communication Server 1

Parameters

`streamName` A string indicating the name of the stream. Use the following syntax:

File format	Syntax	Example
FLV	Specify the stream name as a string, without a filename extension.	`s.play("fileName")`
MP3	Specify the stream name as a string, with prefix `mp3:` or `id3:`, respectively, and without a filename extension.	`s.play("mp3:fileName")` `s.play("id3:fileName")`

MPEG-4-based files (such as F4V, MP4)	Specify the stream name as a string with the prefix `mp4:`. Use a file extension if the file on the server has a file extension. The prefix indicates to the server that the file is in the MPEG-4 Part 12 container format.	`s.play("mp4:fileName")`
		`s.play("mp4:fileName.mp4")`
		`s.play("mp4:fileName.f4v")`

`startTime` A number indicating the playback start time, in seconds. If no value is specified, the value is -2. If `startTime` is -2, the server tries to play a live stream with the name specified in `streamName`. If no live stream is available, the server tries to play a recorded stream with the name specified in `streamName`. If no recorded stream is

found, the server creates a live stream with the name specified in `streamName` and waits for someone to publish to that stream. If `startTime` is -1, the server attempts to play a live stream with the name specified in `streamName` and waits for a publisher if no specified live stream is available. If `startTime` is greater than or equal to 0, the server plays the recorded stream with the name specified in `streamName`, starting from the time given. If no recorded stream is found, the `play()` method is ignored. If a negative value other than -1 is specified, the server interprets it as -2. This parameter is optional.

`length` A number indicating the length of play, in seconds. For a live stream, a value of -1 plays the stream as long as the stream exists. Any positive value plays the stream for the corresponding number of seconds. For a recorded stream, a value of -1 plays the entire file, and a value of 0 returns the first video frame. Any positive number plays the stream for the corresponding number of seconds. By default, the value is -1. This parameter is optional.

`reset` A boolean value, or number, that flushes the playing stream. If `reset` is `false` (0), the server maintains a playlist, and each call to `Stream.play()` is appended to the end of the playlist so that the next play does not start until the previous play finishes. You can use this technique to create a dynamic playlist. If `reset` is `true` (1), any playing stream stops, and the playlist is reset. By default, the value is `true`.

You can also specify a number value of 2 or 3 for the `reset` parameter, which is useful when playing recorded stream files that contain message data. These values are analogous to `false` (0) and `true` (1), respectively: a value of 2 maintains a playlist, and a value of 3 resets the playlist. However, the difference is that specifying either 2 or 3 for `reset` returns all messages in the specified recorded stream at once, rather than at the intervals at which the messages were originally recorded (the default behavior).

`remoteConnection` A NetConnection object that is used to connect to a remote server. If this parameter is provided, the requested stream plays from the remote server. This is an optional parameter.

`virtualKey` A string indicating a key value. Starting with Adobe Media Server 2, stream names are not always unique; you can create multiple streams with the same name, place them in different physical directories, and use the `VirtualDirectory` section and `VirtualKeys` section of the Vhost.xml file to direct clients to the appropriate stream. Because the `Stream.length()` method is not associated with a client, but connects to a stream on the server, you may need to specify a virtual key to identify the correct stream. For more information about keys, see `Client.virtualKey`. This is an optional parameter.

Returns

A boolean value: `true` if the call is accepted by the server; otherwise, `false`. If the server fails to find the stream, or if an error occurs, the `Stream.play()` method can fail. To get information about the `Stream.play()` method, define a `Stream.onStatus()` handler.

If the `streamName` parameter is `false`, the stream stops playing. A boolean value of `true` is returned if the stop succeeds; otherwise, `false`.

Example

The following example shows how streams can be chained between servers:

```
application.myRemoteConn = new NetConnection();
application.myRemoteConn.onStatus = function(info){
    trace("Connection to remote server status " + info.code + "\n");
    // Tell all the clients.
    for (var i = 0; i < application.clients.length; i++){
        application.clients[i].call("onServerStatus", null,
        info.code, info.description);
    }
};
// Use the NetConnection object to connect to a remote server.
application.myRemoteConn.connect(rtmp://movie.com/movieApp);
// Set up the server stream.
application.myStream = Stream.get("foo");
if (application.myStream){
    application.myStream.play("Movie1", 0, -1, true, application.myRemoteConn);
}
```

The following example shows how to use `Stream.play()` as a hub to switch between live streams and recorded streams:

```
// Set up the server stream.
application.myStream = Stream.get("foo");
if (application.myStream){
    // This server stream plays "Live1",
    // "Record1", and "Live2" for 5 seconds each.
    application.myStream.play("Live1", -1, 5);
    application.myStream.play("Record1", 0, 5, false);
    application.myStream.play("Live2", -1, 5, false);
}
```

The following example combines different streams into a recorded stream:

```
// Set up the server stream.
application.myStream = Stream.get("foo");
if (application.myStream){
    // Like the previous example, this server stream
    // plays "Live1", "Record1", and "Live2"
    // for 5 seconds each. But this time,
    // all the data will be recorded to a recorded stream "foo".
    application.myStream.record();
    application.myStream.play("Live1", -1, 5);
    application.myStream.play("Record1", 0, 5, false);
    application.myStream.play("Live2", -1, 5, false);
}
```

The following example calls `Stream.play()` to stop playing the stream `foo`:

```
application.myStream.play(false);
```

The following example creates a playlist of three MP3 files (`beethoven.mp3`, `mozart.mp3`, and `chopin.mp3`) and plays each file in turn over the live stream `foo`:

```
application.myStream = Stream.get("foo");
if(application.myStream) {
    application.myStream.play("mp3:beethoven", 0);
    application.myStream.play("mp3:mozart", 0, false);
    application.myStream.play("mp3:chopin.mp3", 0, false);
    application.myStream.play("mp4:file1.mp4", -1, 5, false);
}
```

The following example plays F4V files:

```
application.myStream = Stream.get("foo");
if(application.myStream) {
    application.myStream.play("mp4:beethoven", 0);
    application.myStream.play("mp4:mozart", 0, false);
}
```

In the following example, data messages in the recorded stream file `log.flv` are returned at the intervals at which they were originally recorded:

```
application.myStream = Stream.get("data");
if (application.myStream) {
    application.myStream.play("log", 0, -1);
}
```

In the following example, data messages in the recorded stream file `log.flv` are returned all at once, rather than at the intervals at which they were originally recorded:

```
application.myStream = Stream.get("data");
if (application.myStream) {
    application.myStream.play("log", 0, -1, 2);
}
```

A server-side stream cannot subscribe to itself. For example, the following code is invalid:

```
// Client-side code
var ns = new NetStream
ns.publish("TestStream");

// Server-side code
st = Stream.get("TestStream");
st.play("TestStream");
```

Stream.playFromGroup()

```
myStream.playFromGroup(ingest)
```

Plays an multicast stream that was ingested into a NetGroup.

For information about ingesting a multicast stream, see `NetGroup.getMulticastStreamIngest()`.

To get information about the `Stream.playFromGroup()` method, define a `Stream.onStatus()` handler.

If the server fails to find the stream, or if an error occurs, the `Stream.playFromGroup()` method can fail.

Availability

Flash Media Server 4.5

Parameters

`ingest` A MulticastStreamIngest object that is ingesting a multicast stream from a source Flash group, or a boolean value of `false` to stop playback from a group.

Returns

A boolean value: `true` if the call is accepted by the server; otherwise, `false`.

If the `ingest` parameter is `false`, the stream stops playing. A boolean value of `true` is returned if the stop succeeds; otherwise, `false`.

Example

The following example shows how streams can be chained between servers:

```
application.myRemoteConn = new NetConnection();
application.myRemoteConn.onStatus = function(info){
    trace("Connection to remote server status " + info.code + "\n");
    // Tell all the clients.
    for (var i = 0; i < application.clients.length; i++){
        application.clients[i].call("onServerStatus", null,
        info.code, info.description);
    }
};
// Use the NetConnection object to connect to a remote server.
application.myRemoteConn.connect(rtmp://movie.com/movieApp);
// Set up the server stream.
application.myStream = Stream.get("foo");
if (application.myStream){
    application.myStream.play("Movie1", 0, -1, true, application.myRemoteConn);
}
```

The following example shows how to use `Stream.play()` as a hub to switch between live streams and recorded streams:

```
// Set up the server stream.
application.myStream = Stream.get("foo");
if (application.myStream){
    // This server stream plays "Live1",
    // "Record1", and "Live2" for 5 seconds each.
    application.myStream.play("Live1", -1, 5);
    application.myStream.play("Record1", 0, 5, false);
    application.myStream.play("Live2", -1, 5, false);
}
```

The following example combines different streams into a recorded stream:

```
// Set up the server stream.
application.myStream = Stream.get("foo");
if (application.myStream){
    // Like the previous example, this server stream
    // plays "Live1", "Record1", and "Live2"
    // for 5 seconds each. But this time,
    // all the data will be recorded to a recorded stream "foo".
    application.myStream.record();
    application.myStream.play("Live1", -1, 5);
    application.myStream.play("Record1", 0, 5, false);
    application.myStream.play("Live2", -1, 5, false);
}
```

The following example calls `Stream.play()` to stop playing the stream `foo`:

```
application.myStream.play(false);
```

The following example creates a playlist of three MP3 files (`beethoven.mp3`, `mozart.mp3`, and `chopin.mp3`) and plays each file in turn over the live stream `foo`:

```
application.myStream = Stream.get("foo");
if(application.myStream) {
    application.myStream.play("mp3:beethoven", 0);
    application.myStream.play("mp3:mozart", 0, false);
    application.myStream.play("mp3:chopin.mp3", 0, false);
    application.myStream.play("mp4:file1.mp4", -1, 5, false);
}
```

The following example plays F4V files:

```
application.myStream = Stream.get("foo");
if(application.myStream) {
    application.myStream.play("mp4:beethoven", 0);
    application.myStream.play("mp4:mozart", 0, false);
}
```

In the following example, data messages in the recorded stream file `log.flv` are returned at the intervals at which they were originally recorded:

```
application.myStream = Stream.get("data");
if (application.myStream) {
    application.myStream.play("log", 0, -1);
}
```

In the following example, data messages in the recorded stream file `log.flv` are returned all at once, rather than at the intervals at which they were originally recorded:

```
application.myStream = Stream.get("data");
if (application.myStream) {
    application.myStream.play("log", 0, -1, 2);
}
```

A server-side stream cannot subscribe to itself. For example, the following code is invalid:

```
// Client-side code
var ns = new NetStream
ns.publish("TestStream");

// Server-side code
st = Stream.get("TestStream");
st.play("TestStream");
```

Stream.publishQueryString

`myStream.publishQueryString`

The query string specified in the stream path when the stream was published.

Use the `Stream.publishQueryString`, `Stream.maxQueueDelay`, and `Stream.maxQueueSize` properties to configure the live queue for live streams. These Server-Side ActionScript properties override the values set in the `Application/StreamManager/Live/Queue/` section of the Application.xml configuration file. The live queue, also known as *live aggregate messages*, batches multiple messages into a single composite message to increase server performance. Dynamic streaming depends on the values of `maxQueueDelay` and `maxQueueSize` to determine when to switch to a higher or lower bitrate stream. Set `maxQueueDelay` to a value long enough to produce a large burst of data.

When you publish a stream, you can specify a query string in the stream path with parameters that specify how to configure the live queue. Access the `publishQueryString` property (for example, from inside the `application.onPublish()` function) to access the query string. Parse the string to get the configuration parameters. Use the values from the configuration parameters to set the `Stream.maxQueueDelay` and `Stream.maxQueueSize` properties.

Note: *Flash Media Live Encoder 3 supports adding query strings to stream names. Earlier versions of Flash Media Encoder did not support query strings.*

Availability

Flash Media Server 3.5

Example

The following client-side code publishes a stream with a query string:

```
ns.publish
("exampleVideo?com.adobe.ams.maxQueueDelay=4000&com.adobe.ams.maxQueueSize=10240");
}
```

The following server-side code gets the query string, extracts the delay and size, and configures the live queue by setting the `maxQueueDelay` and `maxQueueSize` properties:

```
application.onPublish = function(clientObj, streamObj){
    trace("queryString : " + streamObj.publishQueryString);
    // the helper function extracQueryStringArg() is defined below
   delay = extractQueryStringArg(streamObj.pubishQueryString, "com.adobe.ams.maxQueueDelay");
    size = extractQueryStringArg(streamObj.publishQueryString, "com.adobe.ams.maxQueueSize");
    trace("old maxQueueDelay : " + streamObj.maxQueueDelay);
    streamObj.maxQueueDelay = delay;
    trace("new maxQueueDelay : " + streamObj.maxQueueDelay);
    trace("old maxQueueSize : " + streamObj.maxQueueSize);
    streamObj.maxQueueSize = size;
    trace("new maxQueueSize : " + streamObj.maxQueueSize);
}
function extractQueryStringArg(queryString, arg)
{
    var retVal = "";
    temp = arg + "=";
    i = queryString.indexOf(temp);
    if (i != 0)
    {
        temp = "&" + arg + "=";
        i = queryString.indexOf(temp);
    }
    if (i != -1)
    {
        retVal = queryString.substr(i+temp.length);
        i = retVal.indexOf("&");
        if (i != -1)
        {
            retVal = retVal.substr(0, i);
        }
    }
    return retVal;
}
```

See also

`Stream.maxQueueDelay`, `Stream.maxQueueSize`

Stream.receiveAudio

`myStream.receiveAudio()`

A Boolean value that indicates whether the server receives the audio in a stream (`true`) or not (`false`). The default value is `true`.

Availability

Flash Media Server 4.5

Stream.receiveVideo

`myStream.receiveVideo`

A Boolean value that indicates whether the server receives the video in a stream (`true`) or not (`false`). The default value is `true`.

To use Apple HTTP Live Streaming to stream content over a cellular network, Apple requires that one stream be audio-only. To create an audio-only stream, set the `Stream.receiveVideo` property to `false`. The Adobe Media Server livepkgr application (*rootinstall*/applications/livepkgr) includes this code.

Availability

Flash Media Server 4.5

See also

Stream live media (HTTP)

Stream.record()

`myStream.record(flag, [maxDuration, maxSize])`

Records the data passing through a Stream object and creates a file of the recorded stream. You can use this method to do the following:

- Call `Stream.record()` to record a new file or to overwrite the data in an existing file with the recorded data.
- Call `Stream.record("append")` to append the recorded data to the end of an existing file.
- Call `Stream.record("appendWithGap")` to append the recorded data to the end of an existing file, while maintaining the gaps in the stream being recorded. Use this mode only when the `AssumeAbsoluteTime` configuration is `true` and the publisher is sending absolute time. See Using DVR with dynamic streaming.
- Call `Stream.record(false)` to stop recording.

You can record or append in F4V or FLV format. To record a file with multiple codecs, record the file in F4V format.

Before you call the `Stream.record()` method, call the `Stream.get()` method to create a Stream object. The recording format is determined by the filename you pass to the `Stream.get()` method.

Note: *To play or edit F4V files recorded by Flash Media Server in other tools, use the Adobe Flash Media Server F4V Post Processor tool. The tool is available at www.adobe.com/go/fms_tools.*

When you record a stream, the server creates a file with the name you passed to the `Stream.get()` method. The server automatically creates a "streams" directory and subdirectories for each application instance name. If a stream isn't associated with an application instance, it is stored in a subdirectory called "_definst_" (default instance). For example, a stream from the default lecture application instance would be stored here: applications\lectures\streams_definst_. A stream from the monday lectures application instance would be stored here: applications\lectures\streams\monday.

To append a live stream to a file, the stream name passed to the `Stream.get()` method must be different from the live stream name in the client `NetStream.publish()` method. In other cases, the client-side and server-side Stream names can be the same.

Availability

Flash Communication Server 1

Parameters

`flag` One of these values: "`record`", "`append`", "`appendWithGap`", or `false`. If the value is "`record`", the data file is overwritten if it exists. If the value is "`append`" or "`appendWithGap`", the incoming data is appended to the end of the existing file. If the value is `false`, any previous recording stops. The default value is "`record`". For more information about using "`appendWithGap`", see Using DVR with dynamic streaming.

`maxDuration` An optional parameter specifying the maximum duration (in seconds) for a recording. The default value, -1, uses the value of `<MaxDurationCap>` in the Application.xml configuration file. The default value of `<MaxDurationCap>` is -1, which means that there is no maximum duration; the recording length is unlimited.

Set this parameter to -2 or 0 to use the value of `<MaxDuration>` in the Application.xml configuration file as the limit.

`maxSize` An optional parameter specifying the maximum size (in kilobytes) for a recording. The default value, -1, uses the value of `<MaxSizeCap>` in the Application.xml configuration file. The default value of `<MaxSizeCap>` is -1, which means that there is no maximum size; the recording file size is unlimited.

Set this parameter to -2 or 0 to use the value of `<MaxSize>` in the Application.xml configuration file as the limit. The default value is -1.

When a recorded file exceeds the duration or size, the recording stops and when recording stops the `"NetStream.Record.DiskQuotaExceeded"` message and then a `"NetSteam.Record.Stop"` message are sent to the stream's `onStatus` handler.

Returns

A boolean value of `true` if the recording succeeds; otherwise, `false`.

Example

The following example shows a client publishing a live stream, a server recording the stream, and a client subscribing to the recorded stream.

First, the client publishes a live stream:

```
myNetStream.publish("clientStream", "live");
```

Next, the server opens a stream named `serverStream` and stores it in the Stream object `s`. The server-side code plays and records the stream published by the client in F4V format. The name of the recorded file is "serverStream.f4v", which is the name passed to the `Stream.get()` method.

```
//Start recording
s = Stream.get("mp4:serverStream.f4v");
if (s){
    s.record();
    s.play("clientStream");
}
// Stop recording.
s = Stream.get("serverStream");
if (s){
    s.record(false);
}
```

Clients can use client-side code to subscribe to the live stream that was published by the client and recorded on the server:

```
someNetStream.play("mp4:serverStream.f4v");
```

The following example passes the `-1` for `maxDuration` and `maxSize`. This value sets an unlimited duration and size for the recording. These values override the Application.xml values for `<MaxDuration>` and `<MaxSize>`. However the duration and size of the recording cannot exceed the values of `<MaxDurationCap>` and `<MaxSizeCap>` set in the Application.xml file.

```
Stream.record("record", -1, -1);
```

The following example sets `maxDuration` to 50 seconds and uses the `<MaxSize>` setting from Application.xml. The duration and size of the recording cannot exceed the values of `<MaxDurationCap>` and `<MaxSizeCap>` set in the Application.xml file.

```
Stream.record("record",50,-2)
```

See also

`Stream.get()`, `Stream.play()`

Stream.time

```
myStream.time
```

Read-only; the number of seconds the stream has been playing. This value is the timestamp of the latest frame that flowed out of the stream.

Note: *The Stream.time property is limited to 32-bit uint maximum value. This property resets in 49 days.*

Availability

Flash Media Interactive Server 3.5 and Flash Media Development Server 3.5